COMPUTATIONAL INTELLIGENCE

s317661 - Tcaciuc Claudiu Constantin

"The question of whether a computer can think is no more interesting than the question of whether a submarine can swim"

- Dijkstra

AI is a non-compositional compound, Artificial Intelligence is not the predictable sum of the meaning of the component (Artificial + Intelligence).

Computational Intelligence implies acting humanly, more precisely is:
"The art of creating machines that perform functions that require intelligence when performed by people"

- Kurzweil

A machine learning algorithm solves problems by searching, while human learning is based on trial & error and evolution.

Building a machine that follows human learning based implies:

- having access to hereditary material: Structure of the child machine
- benign able to apply mutation: Changes of the child machine
- implementing a natural selection: Judgment of the experimenter

Firstly we need to talk about Evolution, because evolution is not an optimization problem:

- Evolution does not have a goal
- Evolution does not favor strength
- Evolution does not favor intelligence

However, when all variations are accumulated in one specific direction the final outcome may look like the product of an intelligent design.

Problem Solving

Solve By Searching: 6 Friends Problems

- **6 students of Mimetic Learning (3 from computer engineering, 3 from data science), are in a pizzeria to celebrate the end of the COVID pandemic, after the meal they decided to drink a beer in good Irish pub**
- **Unfortunately, the pub is not at walking distance. But one of the student has a tandem bike that can transport two people**
- **Find the fastest way to move all 6 students from the pizzeria (S) to the pub (D), given that**
  - **The tandem bike can transport 1 or 2 students (not 0!)**
  - **If, at any moment, either in the pizzeria or in the pub, the group of students contains more computer engineers (C) than data scientists (D), the computer engineers start to talk about useless details of the hardware of the new GPUs, and the data scientists will run away bored to death — ruining the celebration**
  - **I.e., `count(C) <= count(D) or count(D) == 0`**

Firstly let's define some vocabulary:

- State space
  - A state of possible states that the environment can be in
  - The initial state
  - One or more goal states
- Actions
  - Actions available to the search agent in any given state
  - Actions cost function
  - Transition model
- Path
  - Set of actions (transitions)
  - Optimal solution: minimum cost path from the initial state to a goal state

To solve the problem we also need a good representation of it, for example in this case:

- Possible scenario: (friends in pizzeria, friends in pub) and bike position

So we could use a graph for all possible scenarios (where these scenarios are the nodes), and the edge would be the passage from one scenario to the other.

- (CCCDDD*, -) → (CCDD, CD*) but not (CCDD, CD*) → (CD, CCDD*)

Basically we need to find a solution, these means finding a path between (CCCDDD*, -) and (- , CCCDDD*)

Generate and Test

A possible approach to solve the 6 friends problem is to generate and test random moves: Until everyone is in the pub, move randomly some friends with the tandem and check if the scenario is valid.

In a more general way, to generate and test paths is to use heuristic techniques which guarantees to find a solution if done systematically (and there is a solution):

- Based on depth-first search with backtracking
- trial & error

A good solution generator should be exhaustive, not redundant (should not generate the same solution twice) and informable (absorb information)

Given a complex problem it is often convenient to transform it into something known, or at least more manageable, the transformation can be safe or heuristic.

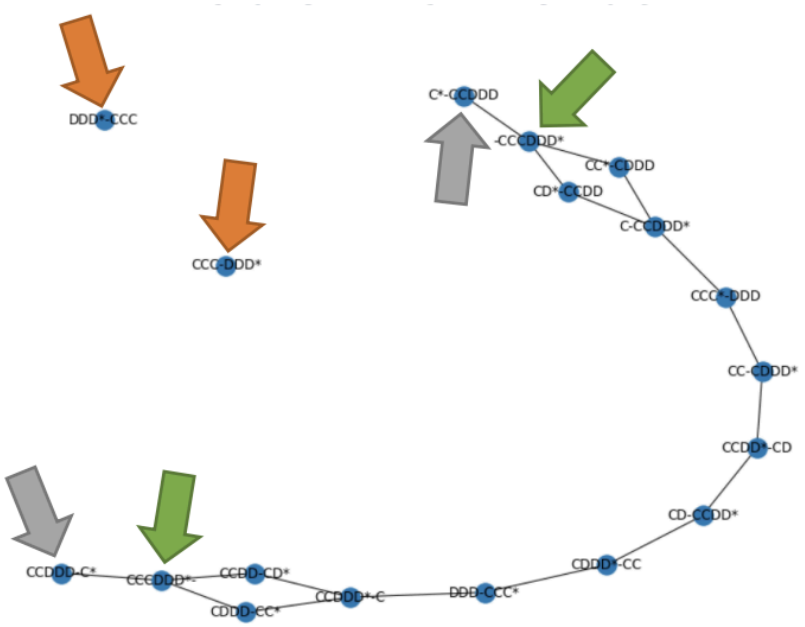
Solve By Searching

Let's now focus on solving the problem, we can define in a basic search problems the same "variables":

- State
- Initial State
- Actions
- Transition Model
- Goal State
- Action Cost

In the case of the "6 friends" problem, we will:

- Enumerate the state space (finite)
- Identify all possible transitions (finite)
- Search for a path from the initial state to the goal state



In general there can be a lot of "agents" type but the more common are:

- Reflex agents (Greedy): simply consider the current situation
- Goal-based agents (Intelligent): maximize the desirability of the future outcome of their action.

But basically problem solving agents look for an action sequence for satisfying the goal, and we are focusing on them.

It's all about "choices", the agent needs to consider the paths with minimum costs that maximizes the reward.

Some terminology:

- Node: state in the (possibly infinite) state space
- Reached: a state for which a node has been generated
- Expand: generate new child (successor) node(s)
- Frontier: set of reached, yet not fully expanded nodes
- Redundant paths: worse ways to reach same state (e.g., cycle)

The problem solving performance can be measured by taking into consideration:

- Completeness: find a solution if there is one
- Cost optimality: find the best solution
- Time complexity: how long does it take?
- Space complexity: how much memory is needed?

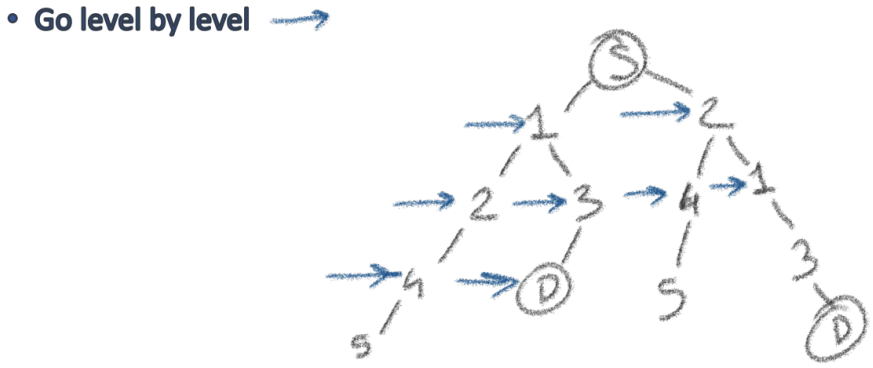
A search algorithm also has to consider the fact that its solution space can also be unsystematic (infinite state space) and also the branching factor (number of successor nodes considered).

The search for a solution can be Uninformative (blind search) or Informative (heuristic search), also there should be no loop solution in the graph of path.

Uninformative search strategies

Breadth-first search

- The root node is expanded, then its successor, then their successors, and so on… (children are added to the frontier with "append")
- Its systematic
- Always complete (if branching factor is less than infinite)
- This is an optimal strategy when all costs are equal

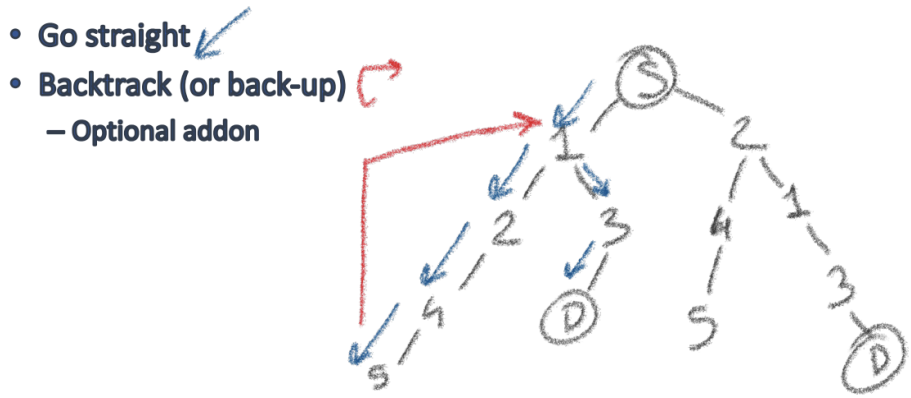


Uniform-cost search (Dijkstra)

- Like breadth-first, but the node with the lowest path cost from the root is expanded (the frontier is a priority queue)
- Breadth-first can be seen as Dijkstra with uniform cost

Depth-first

- The deepest node in the frontier is expanded (children are added to the frontier with "appendleft")
- Problems of cycles



Beam search

- Breadth-first + limited number of nodes in each level (children are added to the frontier with "append", then "trim" drop a certain number of nodes from the frontier)
- Not complete

Depth-limited search

- Depth-first + limited diameter (children are added to the frontier with "appendleft", and deep nodes are discarded by "trim")
- Not complete unless iterative deepening is also used

Bi-directional search

- Search forward from initial state; backward, from goal

Informative search strategies

Greedy best-first

- Expand the node with the maximum expected value
- Not optimal
- Incomplete (unless backtracking)

A* search

- Best-first using function: f(n) = g(n) + h(n)
- g → is the actual cost
- h → is the estimated cost (heuristic)

A* is a complete and optimally efficient strategy, that will always find a solution (not the best, but a good one), of course under reasonable assumptions

- It always computes the path with minimum cost by expanding a minimum number of nodes.

To be able to find a solution we need to have h(n) to be:

- Admissible: if it never overestimates the cost to reach the destination node (in a minimization problem). If the search space is a tree, admissible heuristic guarantee optimal solutions
- Consistent: if for every node n for every successor node ns of n:
    - h is monotone
    - consistent is always heuristic, because it guarantees optimal solutions (once a node has been expanded it won't be reached again at a lower cost)
- If h is consistent then is admissible.

Caveats:

- "Usually" when h is admissible it is also consistent
- Any optimistic heuristic is admissible
- To create admissible heuristic → simplify the problem by relaxing constraints

Space/Time complexity O(b^d)

- (branching factor, depth of optimal solution)

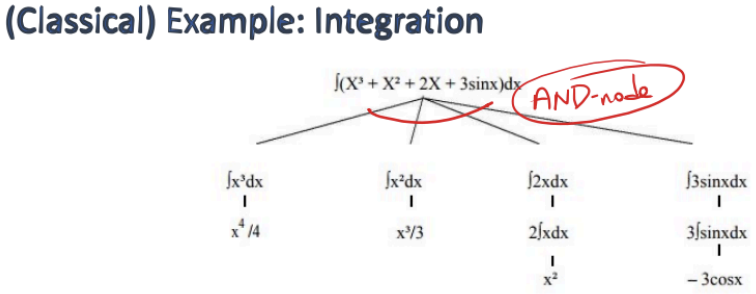
Other problem solving techniques

AND/OR Search Trees

Search Trees:

- The agent must perform both action A and B
- The agent can perform either action A or B

Given a large (complex) problem it is often convenient to decompose it in smaller sub-problems



Constraint Satisfaction Problem (CSP)

Factored representation

- Set of variables, each one has a value

Backtracking algorithm

- Recursive, global state
- Several possible improvements

Goal Tree
The complexity of the goal tree is a consequence of the complexity of the problem, not of the algorithm.
But a general rule:
- The complexity of the behavior is the maximum between the complexity of the program and the complexity of the environment

(non ho capito molto bene)

Single State Methods

Single-state local search

Sometimes we are only interested in the solution, and the path taken is irrelevant (e.g, Sudoku) or trivial given the solution

The process is something like this:
- Start with a state (possibly illegal)
- Iteratively improve/patch it searching in neighboring states

So basically the two methods are to boldly go towards the goal vs. Wandering around in search of a solution.

Hill Climbing

The basic strategy of hill climbing is: search for a solution or solve optimization problems.
It's closely related to gradient ascent, but it doesn't require one. Iteratively test new candidate solutions in the region of a candidate solution (neighborhood).

A solution is taken only if they are an improvement, so basically quickly climb up to the first local optimum.

```
S <- some initial candidate solution
repeat
      r <- Tweak(Copy(S))
      if Quality(R) > Quality(A) then
      S <- R
until S in the ideal solution or we have run out of time
```

```
return S
```

The two basic implementation to achieve improvements are:

- First-improvement Hill Climber (Random-Mutation) - RMHC
- Steepest-step Hill Climber (also called Steepest Ascent) - SAHC

To tweak one state there are some things we can do:

- Use larger sample (directly favor exploitation)
- Occasionally makes larger, more random changes (directly favor exploration)
- Accept worsening changes hoping to escape local optima (simulated annealing)
- Restart from scratch from a state (always a good idea, if you have spare time)

The termination state or Stopping condition could be (this is probably valid for everything):

- Best solution found
- Total number of evaluations (best baseline to evaluate an algorithm)
- Total number of steps (useful in parallel environments)
- Wall-clock time
- Steady-state (give up if chance of improvement is low)

To bypass the problem of local maxima solutions the basic idea is to accept a worsening solution, but only if f(s) > f(s'). This is called Simulated Annealing

- **Basic idea: Hill Climbing with p≠0 of accepting a worsening solution s' where $f(s) > f(s')$:**

$$p = e^{-\frac{f(s)-f(s')}{t}}$$

- **Where: s is the current solution, s' is the tweaked one, and t is the *temperature***

Tabu search

Usually this is implemented as a variation on SACH w/Replacement, and involves generating n new solutions, but discarding the tabu ones. The caveat is that it works only in discrete spaces.

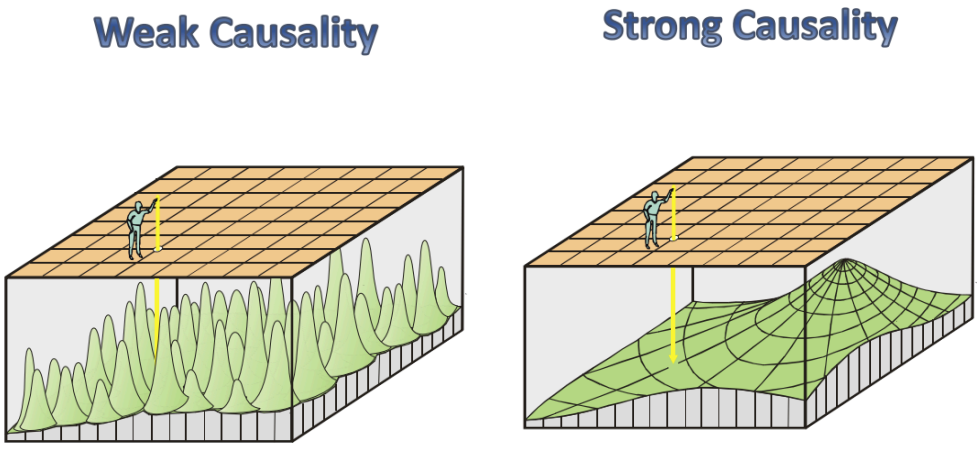
Iterated Local Search

It's a clever version of HC with Random restarts, where the process restarts from the point selected by `<< new_starting_point(global, last) >>`, and it works almost as "black magic", but simply returning the best solution found so far usually works.

(1+1) (1+λ) (1, λ) - Evolution Strategies

Those are technically not HC, but "Evolution Strategies". The solution is a list of real numbers.

- (1+1): A first-improvement HC were tweak is performed using a Gaussian mutation on each element
  - $x1' = x1 + N(0, \sigma)$
  - "σ" is the mutation step
- (1+λ): A steepest-step HC where λ is the sample size, and the tweak is performed using a Gaussian mutation as in (1, 1)
  - Also called plus strategy
- (1, λ): same as (1+λ) but the current solution is always replaced, even if all the new λ solutions are worsening.
  - Also called comma strategy
  - Note: using large λ reduces the risks



The best mutation step σ can be selected by:

- Random guess
- Experience
- Meta-optimization

Dynamic Strategy

- Monitor the number of successor over a given amount of steps (era)
- Tweak σ to balance exploration and exploitation

- Endogenous parameter
- Problem-space vs algorithm-space

Self-adaptation
A self adaptive strategy implies adding σ to the list of parameters, than mutate σ and v in different steps (the normal convention is to mutate firstly σ then vi)

- <v1, …, vn, σ>

Another self adaptive strategy implies adding σ for each v, and the rest is the same.

- Two learning rates.
- <v1, …, vn, σ1, …, σn>

Population Methods

Evolutionary Computation

A (μ, λ) evolutionary strategies implies having a population of μ in individuals

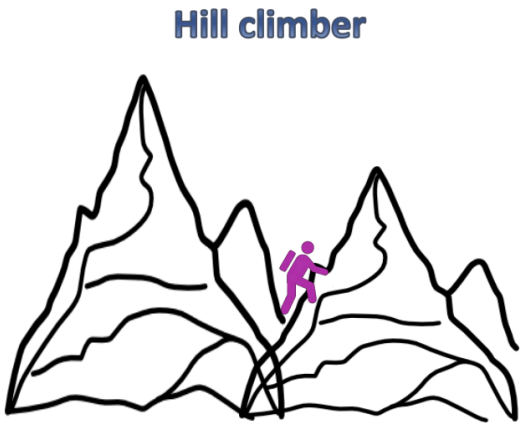
- Self-adaptive strategy: parameter values + mutation step
- Where in each generation:
  - Generate λ new individuals by mutating random parents
  - Select the best μ out of the λ for the next generation

Evolution is the accumulation of tiny variations, it's basically a sequence of two contributions:

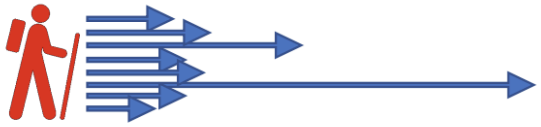
- variations (mostly random)
- selection (mostly deterministic)

In evolution, changes are not designed but merely evaluated, also is not a random process though random variability "afford materials"

Let's have some example:



Self-adaptive variable step size



Population based

So let's focus on evolution and define it, as we said before:

- Evolution does not have a goal
- Evolution does not favor strength
- Evolution does not favor intelligence

However, when all variations are accumulated in one specific direction the final outcome may look like the product of an intelligent design.

Stated that we can define Evolutionary algorithm as: "A generic population-based metaheuristic optimization algorithm that uses some mechanism vaguely inspired by biological evolution".

A bit of terminology:

- Candidate solution -> individual (encode a potential solution for the problem)
- Set of candidate solutions -> population
- Ability to solve the problem -> fitness
- Sequence of steps -> Generations

An Evolutionary algorithm implies some structural things to do:

- Creation of the initial population (almost always randomly)

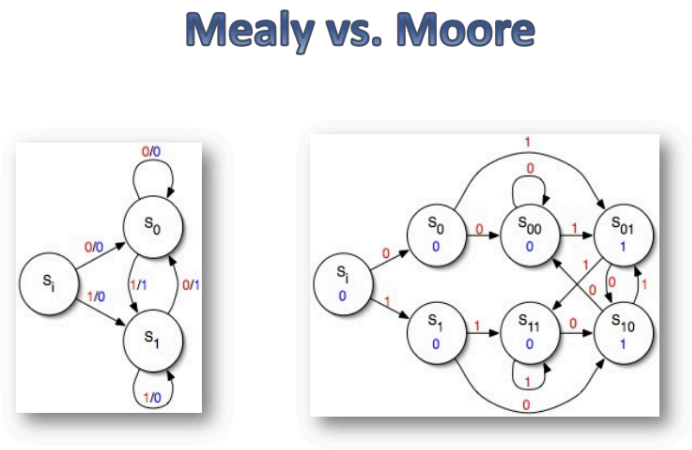
- Parent Selection (select the parents that will generate the offspring, it almost always based on fitness)
- Creation of the offspring (newly generation are created)
- Evaluation (the remaining individual are evaluated)
- Survival selection (only the best individual are selected)

Evolutionary Programming

When we talk about evolutionary programming we are basically talking about an optimization algorithm that focuses on solving specific problem domains by adapting the structure of candidate solutions over generations.

Solutions are typically represented as arrays of parameters or structures that are specific to the problem at hand. EP often emphasizes the self-adaptation of parameters during the evolutionary process. This means that the algorithm can dynamically adjust its parameters based on the success or performance of the solutions.

FSM
A finite state machine implies:
- S states
- I inputs
- O outputs
- δ: SxI -> SxO



Canonical EP
In the canonical EP often there is no parent selection and no recombination:
- One point in the search space stands for a species, not for an individual and there can be no crossover between species.

So when a mutation is taking place:
- we add a state, then remove a state

- change transition, change initial state
- change output

The survival selection is similar to (μ + μ) ES
- q random matches, the best μ survive
- Selection pressure: f(q)

The stopping condition is often "Steady state" to reduce computational power, and not have useless computation. Also there is no elitism.

Genetic Algorithm

Best known as evolutionary algorithms "GAs", their original goal was to study adaptation in natural and artificial systems. They are an (over) simplification of biological processes.

In brief:
- Solutions are typically encoded as strings of parameters (chromosomes), and the genetic operators include crossover and mutation.
- Crossover Involves recombination or exchange of genetic information between parent solutions. Mutation Introduces small random changes to individual solutions.
- "GAs" use selection mechanisms like roulette wheel selection or tournament selection to choose individuals for reproduction (is fully deterministic, meaning that it is based on fitness).

An individual can be encode as:
- Classical encodings
    - Bit strings
    - Integers
    - Real numbers
    - Permutation
    - …
- Unique representation
- Infeasible solutions

The creation of the offspring implies that the offspring must inherit the main characteristics of the parents:
- No gender distinction
- Genetic operators: Recombination & Mutation

Alternative Types of GA

Strategy 1:

- Selection: Copy fittest individual into new generation
- Mutation rate: Individual can be slightly modified during the copy
- Crossover rate: Recombination

Strategy 2:

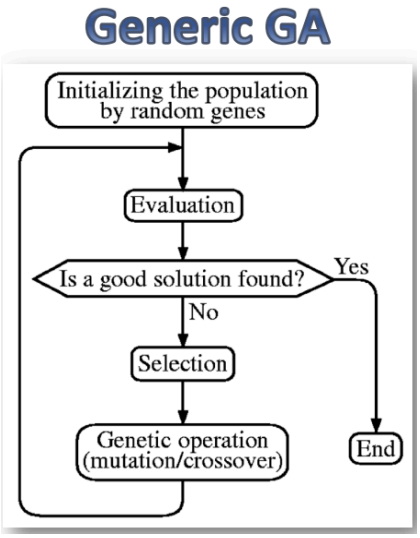
- Selection: Choose parents
- Apply genetic operators
- Mutation rate: The offspring can be slightly modified after

Also using a steady state technique to reduce the computational time is a good thing.

Potential Problems

The potential problems are that we may end up with infeasible/invalid solutions, in that case we can:

- Discard the solutions
- Heuristic repair
- Accept invalid solution with penalty



GA vs ES

GA and ES are both Evolutionary Algorithms quite similar in my opinion and the choice between one and the other depends on the specific characteristics of the optimization problem, including the nature of the solution space, the dimensionality of the problem, and the balance between exploration and exploitation needed for effective optimization.

Comparison:

- Exploration vs. Exploitation: GAs are generally better at global exploration of the search space, making them suitable for problems with diverse solutions. ES, on the other hand, is often better at local exploration and fine-tuning, making it suitable for problems where precision matters in a specific region.

- Representation: GAs explicitly use chromosomes, and crossover plays a significant role in combining genetic information. In ES, the focus is on the direct manipulation of real-valued vectors, and mutation is more prominent.

- Adaptation: ES typically emphasizes the self-adaptation of parameters, allowing it to dynamically adjust to the characteristics of the optimization landscape.

Genetic Programming

Genetic Programming is designed for evolving computer programs or expressions. It is used for symbolic regression and evolving algorithms.

Solutions are represented as tree structures, often representing mathematical expressions or computer programs.
- Crossover: Involves exchanging subtrees between parent solutions.
- Mutation: Involves altering subtrees or nodes in the tree structure.

GP is applied to problems where the goal is to evolve a symbolic expression or program, such as evolving control strategies or mathematical models.

GP vs ES/EP
- Non linear structure
- Variable number of loci
- Both width and depth

Survival
The survival strategy evolved during the years:
- Traditionally:
    - Purely generational approach
    - large populations
- More recently:
    - Elitism
    - Steady state

- Brood recombination

Over Selection

- Population is split in 2 groups (G1: best x%, G2: everyone else)
- 80% of the parent from G1, 20% from G2

Fuzzy Logic

Long story short: operation on fuzzy set A and B

- NOT A -> 1-A
- A AND B -> MIN(A, B)
- A OR B -> MAX(A, B)

We are basically talking about uncertain vs imprecise information, in a problem the:

- probability describes the uncertainty of an event (whether an event occurs is random)
- fuzziness describes event vagueness (to what degree it occurs is fuzzy)

Knowledge

A knowledge-based agent includes a knowledge base and an inference system:

- A knowledge base is a set of representation of facts of the world
- Each individual representation is called a sentence
- The sentences are expressed in a formal language

The agent operates as follows:

- It TELLs the knowledge base what it perceives
- it ASKs the knowledge base what action it should perform
- It performs the chosen action

Entailment

Entailment means that one things follows from another: KB $\vdash \alpha$

- Knowledge base KB entails sentence α if and only if α is true in all worlds where KB is true

Inference, Soundness, Completeness

- KB $\vdash_i \alpha$ sentence α can be derived from KB by procedure *i*
  - Soundness: *i* is sound if whenever KB $\vdash_i \alpha$, it is also true that KB $\models \alpha$
  - Completeness: *i* is complete if whenever KB $\models \alpha$, it is also true that KB $\vdash_i \alpha$
  - Preview: we need a first-order logic which is expressive enough to say almost anything of interest, and for which there exists a sound and complete inference procedure. That is, the procedure will answer any question whose answer follows from what is known by the KB.

Knowledge Representation

The object of Knowledge Representation is to express knowledge in a computer-tractable form, so that agents can perform well.

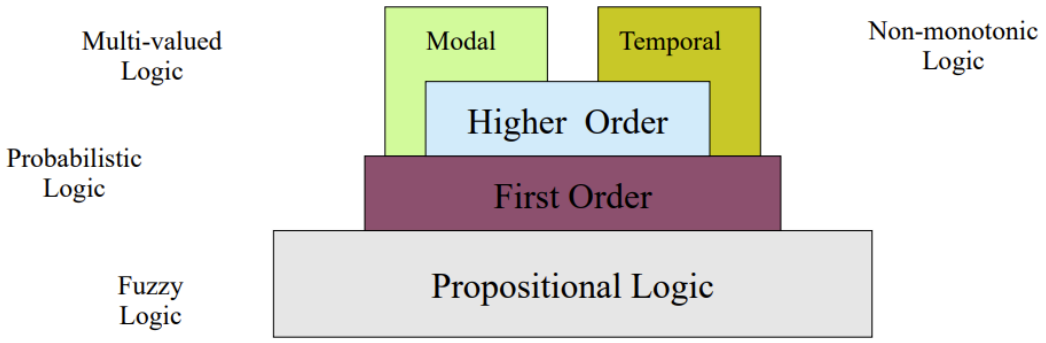
A Knowledge Representation language is defined by:

- syntax: possible sequence of symbols that constitute sentences of the language
- semantics: the facts in the world to which the sentences refer

Semantics maps sentences in logic to facts in the world. The property of one fact following from another is mirrored by the property of one sentence being entailed by another.

- A sound inference method derives only entailed sentences
- Analogous to the property of completeness in search, a complete inference method can derive any sentence that is entailed

Logic as a KR language



Propositional logic

- Logical constants: true, false
- Propositional symbols: P, Q, S, ... (atomic sentences)
- Wrapping parentheses: (...)
- Sentences are combined by connectives:
  - ∧ and [conjunction]
  - ∨ or [disjunction]
  - ⇒ implies [implication / conditional]
  - ⇔ is equivalent [biconditional]
  - ¬ not [negation]
- Literal: atomic sentence or negated atomic sentence
  P, ¬ P

A simple language useful for showing key ideas and definitions, the user defines a set of propositional symbols, like P and Q, and defines the semantics of each propositional symbols:

- P means "it is hot"
- Q means "it is humid"
- R means "it is raining"

A sentence (well formed formula) is defined as follows:

- A symbol is a sentence
- if S is a sentence, then ¬S is a sentence
- if S is a sentence, then (S) is a sentence
- if S and T are sentences, then (S v T), (S □ T), (S →T) and (S ↔ T) are sentences
- A sentence results from a finite number of applications of the above rules

Logical inference is used to create new sentences that logically follow from a given set of predicate calculus sentences (KB).

- An inference rule is sound if every sentence X produced by an inference rule operating on a KB logically follows from the KB (the inference rule does not create any contradictions)
- An inference rule is complete if it is able to produce every expression that logically follows from (is entailed by) the KB.

- Here are some examples of sound rules of inference
  - A rule is sound if its conclusion is true whenever the premise is true
- Each can be shown to be sound using a truth table

RULE	PREMISE	CONCLUSION
– Modus Ponens	$A, A \rightarrow B$	B
– And Introduction	A, B	$A \wedge B$
– And Elimination	$A \wedge B$	A
– Double Negation	$\neg\neg A$	A
– Unit Resolution	$A \vee B, \neg B$	A
– Resolution	$A \vee B, \neg B \vee C$	$A \vee C$

First Order Logic (FOL)

FOL models the world in term of:

- Objects, which are things with individual identities
- Properties of objects that distinguish them from other objects
- Relations that hold among sets of objects
- Functions, which are a subset of relations where there is only one "value" for any given "input"

The user provides:

- Constant symbols, which represent individuals in the world
- Function symbols, which map individuals to individuals
- Predicate Symbols, which map individuals to truth values

FOL provides:

- Variable symbols
- Connectives
- Quantifiers

A term (denoting a real-world individual) is a constant symbol, a variable symbol, or an n-place function of n terms.
An atomic sentence (which has true or false value) is an n-place predicate of n terms, A complex sentence is formed from atomic sentences connected by the logical connectives.
A quantitative sentence adds quantifiers $\forall$ and $\exists$, and a well-formed formula is a sentence containing no "free" variables (that is, all variables are "bound" by universal or existential quantifiers).

High order Logic (HOL)

HOL allows quantification over relations, instead FOL only allows quantification over variables, and variables can only range over objects.

Multiple Agents

?

Test problems and Fitness

Fitness

Fitness is the quantitative representation of natural and sexual selection with evolutionary biology, and is basically the individual reproductive success.

In computer science the fitness is the ability to solve the given problem:

- The bigger the better
- Optimization: "maximize the fitness"

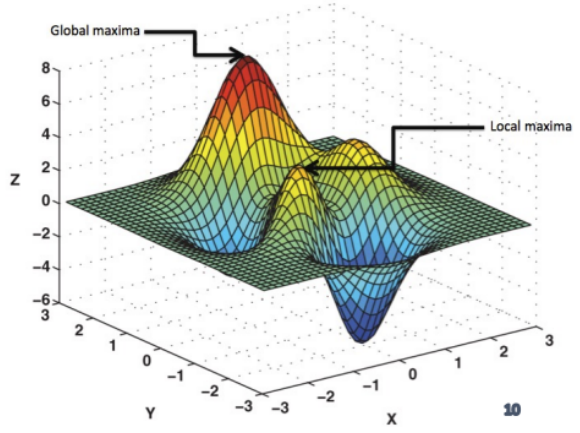
It is often artificial, real value of the solution and arbitrary penalties trying to direct the search into specific directions

How to define the fitness function f(x) in order to minimize the result of the evaluation E(x)?

- f(x) = - E(x) some algorithms may not work with negative fitness values.
- f(x) = K - E(x) some algorithms may be impaired by offsets.
- f(x) = 1 / E(x) the difference of fitness values does not reflect the difference of evaluations.

Fitness Landscape:

- Unimodal: only local optima exists which is also the global optimum
- Multimodal: many local optima exist, one or more are global optima



Exploration vs Exploitation:

Multi-Armed Bandit Problem: Multiple levers to pull, each with different payout

percentages

- Try new levers to see which one works best: exploration
- Keep pulling the lever with the best payout: exploitation

Classic Benchmarks

One Max

- Goal: build a string of ones
- Fitness: sum of the ones
- The basic form are its variants are still among the most used benchmarks in EA

[next set of slides are not related?????]

Representation

- Fitness: how well the candidate solution is able to solve the target problem
- Genotype: the internal representation of the individual, i.e., what is directly manipulated by genetic operators
- Phenotype: the candidate solution that is encoded in the genotype
    - the intermediate form in which the genotype needs to be transformed into for evaluating fitness
    - if genotype can be directly evaluated: genotype and phenotype coincide

Encoding:

Individuals (phenotypes) are encoded into chromosomes (genotypes). The fitness landscape shows the solution, not the problem space.

Small differences should bring small differences in fitness (locality).

Genetic Operators:

- Mutation:
    - Single parent
    - Small changes
    - May have a variable strength
    - May be repeated
- Recombination (Crossover)
    - Two or more parents
    - The offspring must inherit traits from all parents
    - Associated with the idea of exploration

In general, it’s better to have both mutation and recombination operators

(mutation: small tweaks exploitation, recombination: big changes, exploration)

- Mutation only could be fine
- Recombination only would not work

Binary Representation:

Virtually everything can be encoded as a binary number

- Some encoding destroy the structure of the solution
- Some are more reasonable

The mutation is essentially a bitflip, and the recombination is almost always a 1-cut crossover (can be exploited if we know about the structure of our problem, but this is not usually the case).

- The n-cut crossover is a generalization of 1-cut crossover.

Integer Representation:

Mutation:

- random resetting (the allele in a random locus is replaced with a random value)
- creep mutation (the allele in a random locus is slightly increased or decreased)

Recombination (same as binary representation)

- 1-cut crossover
- n-cut crossover
- uniform crossover

Permutation Representation:

This type of representation can be seen as a quite special type of integer representation:

- the information is stored in the absolute position
- the information is stored in the order
- the information is stored in the sequence

Mutation:

- Swap mutation (the allele in two random loci are swapped)
- Scramble mutation (some alleles are randomly selected, loci are shuffled - the number of loci is the strength of the mutation)
- Insert mutation (two loci are randomly selected and the allele in the second moved next to the one in the first, shifting along to make room)
- Inversion mutation (two loci are randomly selected and all the alleles between them are inverted)

Recombination:

- Cycle crossover (choose two loci l1 and l2 and copy the segment between them from p1 - copy the remaining unused values)
- Partially mapped crossover (choose two loci l1 and l2 and copy the segment between them from p1 to o1 - copy missing alleles occupying the segment in checking their positions in p1 and p2)
- Inver over (asymmetric crossover, get only one trait from the second parent)

Floating-Point Representation:

Mutation: Gaussian mutation (same as always)

Recombination:

- Uniform crossover (discrete)
- Intermediate crossover
- Averaging crossover

[didn’t really understand this last part]

End