SEARCH STRATEGIES

When we search for a solution in the SEARCH SPACE we are indeed trying to find a sequence of ACTIONS that bring an INITIAL STATE to a GOAL STATE.

Building a SEARCH TREE is intrinsically EXPONENTIAL

child is the result of applying an action
Leaves are goals

We can find solution in a "general" way (UNINFORMED STRATEGY) or we can exploit human knowledge on the domain to reduce the complexity on the branching factor (INFORMED SEARCH).

Finding a solution is done trough a SEARCH STRATEGY which is:

COMPLETE → we find a solution
in
OPTIMAL → we find THE BEST solution.

A SEARCH STRATEGY is which nodes need to be opened first (and why) to easily reach (possibly an optimal) goal.

To implement a SEARCH STRATEGY we follow a a general algorithm

```
GENERAL-SEARCH ( problem, QUEUINGFN ):

    nodes = [Node ( problem.initial_state)]
    while TRUE:
        if nodes is empty: return FAIL

        node = nodes.pop()
        if node.IS-GOAL: return node
        nodes = QUEUINGFN ( nodes, node.expand() )
```

In this way by defining how nodes are queued we implement a SEARCH STRATEGY as we have control on which node to expand next.

UNINFORMED SEARCH STRATEGIES

BREADTH-FIRST : Used on small problems but it's the foundation of other strategies

Expand the best deep node

↓

FRINGE is a QUEUE (FIFO)

↙

SET OF OPEN NODES

↓

at the end

FRINGE = GOAL

Worst case scenario : $O(b^d)$ in space and time if goal is rightmost solution at depth d.

(b: actions per node, d: max depth)

It's COMPLETE however not OPTIMAL in case not all actions have the same cost in solution context.

UNIFORM COST : Same as breadth first but expand some-depth nodes in increasing (known) cost function.

DEPTH FIRST : Expand deeper nodes first. (at equal depth usually the leftmost)

↓

Proceed expanding a child's children until a leaf is found

It is linear in space $O(bd)$ and similar to BREADTH FIRST in time ($O(b^d)$ in worst case, consider the same example)

We just need a STACK (LIFO) to handle the data because only one path at time is considered.

NOT COMPLETE if we find loops.

LIMITED DEPTH SEARCH : Depth first but we stop at a specified depth and choose another node to expand (if available)

NOT COMPLETE as we could stop before finding a goal but we avoid loops.

For " MAX DEPTH $\to +\infty$ => DEPTH FIRST
MAXDEPTH = 0 => BREADTH FIRST

ITERATIVE DEEPENING : Limited depth with increasing max depth.
In general the best strategy to use as repeated nodes expansion does not worsen significantly the complexity.

We combine the best of DF (low memory complexity) with the best of BF (optimal) compromising w\ time complexity.

INFORMED STRATEGIES

Heuristics are used to improve algorithms performances both in space and time.
Those improvements however need to reduce significantly the complexity otherwise they just become a layer of confusion over the strategies.

BEST FIRST: The most promising child is chosen trough an evaluation function.

It's not optimal because the technique always chooses the most promising child regardless of how deep it is. By following this kind of procedure long path to the goal are taken if the evaluation of each node in the path is low.
We should however consider how far we have gone and if it would be better to backtrack and choose a most promising node.

This approach is also known as GREEDY.

In worst case scenario time and space are $O(b^d)$ like BREADTH-FIRST but w\ a **GOOD** heuristic we can actually improve results.

A* ALGORITHM: Similar to GREEDY BEST FIRST, but we actually also consider the reached depth to prevent long promise looking paths.

$$f(n) = g(n) + h'(n)$$

- $f(n)$: evaluation function
- $g(n)$: reached depth
- $h'(n)$: heuristic function

We are trying to combine depth first efficiency w\ uniform cost completeness and optimality.

$h'(n)$ IS FEASIBLE => A* IS OPTIMISTIC

$h'(n) \leq h(n)$ (true distance to goal)

e.g. $h'(n)$ need to always underestimate the distance.

→ if $h'(n) = 0$ => BREADTH-FIRST

Choosing an heuristic is usually hard. We prefer bigger value heuristics

Sometimes relaxed constraints heuristics are still good

$h'_1(n) < h'_2(n) \Rightarrow h'_2(n)$ is BETTER

or we can use $h'(n) = \max\{h'_1(n) \dots h'_m\}$

GRAPH SEARCH: W\ A* — W\ trees we always assumed we couldn't bump into a node more than one time → WRONG!

We can consider our search space as a GRAPH: if a node is traversed twice we are going into a loop.

↓

The only way to solve this is to keep track of the visited nodes.

$h'(n)$ IS CONSISTENT (MONOTONE) => A* FOR GRAPHS IS OPTIMAL.

↓

$h'(n) = 0$ if n is GOAL

$h'(n) \leq c(n, a, n') + h'(n')$ $\forall$ n in action space, $n' \in$ children(n)

cost from n to n' through a

LOCAL SEARCH

Start from a solution and improve it by moving into NEIGHBOUR solutions.

↓

Path can't be important but only the result should be considered (n queens problem, TSP ...)

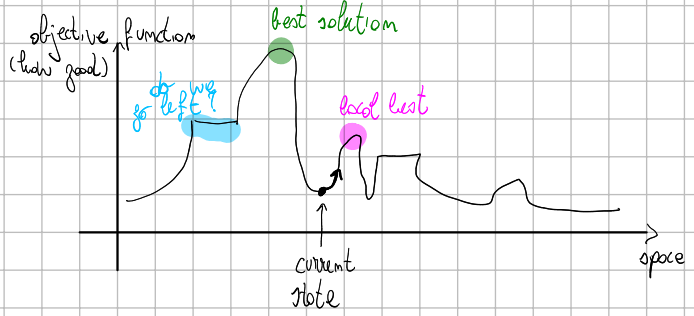
HILL CLIMBING: Perform a move if brings to a better state of the NEIGHBOROOS => find a LOCAL MINIMUM

↓

there could be a better solution as we stop as soon as we can't find anything better in the neighborood

We could use bigger neighborood to get higher chances of having a nearby global min (or max) but obv. the complexity grows.

Depending on the LANDSCAPE the optimization problem becomes harder



A way of escaping from flat areas and local min is needed in order to found the best solution.

↓

exploit META HEURISTICS to search in a more effective way.

↓

ex. accept non improving moves
change neighborood evaluation function during search

- SIMULATE ANNEALING: Allow worsening moves in the first iterations and gradually use only refining moves

↓

associate to each action a probability of being taken

$$p(s, a, s') = e^{-\frac{f(s') - f(s)}{T}}$$

↓ going from s to s' trough a

where T (temperature) is how fast worsening moves probabiliti decays

- $T_{k+1} = \frac{\Gamma}{\log(k + k_0)}$ → LOGARITHMIC, converges to optimal but in ∞ iterations
- $T_{k+1} = \alpha T_k$, $\alpha \in (0, 1)$ → GEOMETRIC
- Non Monotonic T → T increases and decreases to exploit intensification (staying in the neighborood) or exploration of new neighb r

- TABU SEARCH: Keep track of recent states and forbid them.

↓

More memory needed → store moves instead of states

↓

forbidding moves could prevent us from moving to a desired direction

← use an ASPIRATION CRITERIA

↓

forbidden moves are allowed if they bring to a better state.

- ITERATED LS: ① INTENSIFICATE: Reach local optima trough hill climbing
② PERTURBATE: Escape from local optima by exploiting history

Each method is stopped when a goal criterion is satisfied **OR** when a timeout is expired.

Sometimes the landscape could be so complex (lot of local minima) that previous techniques take too much time to converge.

POPULATION BASED METAHEURISTICS can be used to simulate genetic evolution in keeping the good parts and evolving the worst parts.

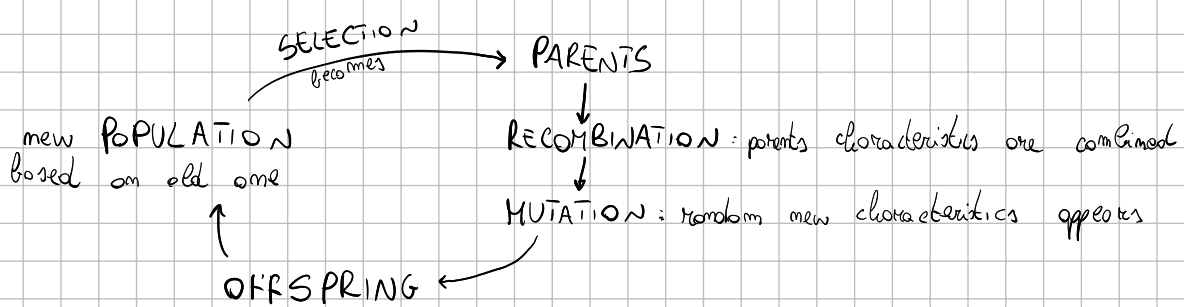
GENETIC ALGORITHMS are based on EVOLUTION:

ADAPTATION: organisms suits the habitat
INHERITANCE: children gains parents traits
NATURAL SELECTION: Only those w\ good traits emerges while the others are extingued if evolution doesn't adapt them.

↓

The FITEST individuals have long life
Their children are similar but different: only those whose difference better suits the habitat.

In AI context individual → possible solution (also genotypes)
habitat → problem
fitness → solution quality



Genotype can be seen as a binary vector, with RECOMBINATION we build a new vector using two vectors, with MUTATION we randomly flip some bits.

During OFFSPRING only a selection of the population survives:

- PROPORTIONAL SELECTION: the fittest the more individuals survives
- GENERETIONAL REPLACEMENT: all childrens become population BUT good traits could go lost if noone have inherited them.

↓

BEST N: keep the best of children and parents

We can implement genetic operators w\ the use of linear combination

$x_c = \lambda_1 (x_p + \delta) + \lambda_2 (y_p + \delta)$ where x_c is child value, δ is random (mutation) and λ_1, λ_2 rule how much of parents x_p, y_p goes into x_c

We can combine LOCAL SEARCH and GENETIC ALGORITHMS by performing a local search on each individual to obtain the best possible outcome in its neighborhood

↓

MEMETIC ALGORITHMS

SWARM INTELLIGENCE

By using this search technique problem size isn't actually important as solutions are only computed locally by "cooperating" agents. The technique is thus robust and efficient if used w\ DISTRIBUTED COMPUTING.

Self adjusting colonies that learns to find the best solution by collectively searching for a result.

Agents are very simple creatures which act NON DETERMINISTICALLY by performing action and observing feedback.

↓

e.g communicate w\ other agents

↙ DIRECTLY

↘ INDERECTLY (STIGMERGY)
By changing the environment

If we remove agents from the colony the remaining agents will keep seeking for a goal

↓

GRACEFUL DEGRADATION

ANT COLONY

Ants follow path to food w\ high pheromone trails, left by other ants who passed there

↓

they tend to find the optimal path as only those who found food will come back to deposit it (thus leaving more pheromone).

Pheromone model is a PROBABILISTIC PARAMETRIZED MODEL

Each connection in the graph have 2 associated informations: PHEROMONE τ
HEURISTIC η

τ encodes the colony long term memory

The problem solution is represented by building a graph in which arcs are explored arcs of the problem graph and the pheromone model dictate which arc is better.

ACO - ALGORITHM (first proposed)

Each ant search his best solution, at the end pheromone is "spread" on

$$p_{ij} = \frac{\tau_{ij}^{\alpha} \eta_{ij}^{\beta}}{\sum_{k} \tau_{ik}^{\alpha} \eta_{ij}^{\beta}}$$

p_{ij} ↓ probability of choosing arc ij

α, β parameters on PHEROMONE vs HEURISTIC importance

$\eta = \frac{1}{d_{ij}}$ → distance to goal

each ant will choose an arc based on how much it wants to follow the group (τ) and how much it wants to take initiative (η).

$$\tau_{ij} = (1 - \rho)\,\tau_{ij} + \sum_{k=1}^{m} \underbrace{\Delta \tau_{ij}^{k}}_{\text{''}}$$

ρ ↓ evaporation of pheromone coefficient

$\frac{1}{L_k}$ if ant_k used arc ij else 0

L_k ↓ length of the followed path

Delayed pheromone spread is used so that so that L_k is known.

The ant colony doesn't find deterministically the solution but, trough reinforcing and updating τ a suboptimal solution is found and refined.

The nondeterministic approach is FUNDAMENTAL in making the algorithm work otherwise ant would never refine the solution and only use the first found path.

↓

this could led to slow convergence ⇒ DEMON ACTIONS are taken

↓

local search each solution and leave additional pheromone to guide the ants over a preferred solution.

HONEY BEE-COLONY

Agents have different roles:

- scouts → discover food
- employed → recognize nectar source
- onlookers → observe employed and choose particular nectar.

① Each bee start from a random position in the graph, uniformly distributed.

② Employed bees locally search for new food and shares the outcome

③ On looker bees observe employed bees informations and join the most promizing one.

④ Scouts search for new food sources

If an employed doesn't find food becomes a scout

At termination the food source with more bees is the best solution.

PARTICLE SWARM OPTIMIZATION

Overall swarm has a common goal, each agent can do:

INDIVIDUALISTIC CHOICE
leave the flock

SOCIAL CHOICE
follow flock

Each agent communicate the result to the flock which will imitate better solutions

Each agent is part of more SUBGROUP ∈ FLOCK. By sharing informations with its group the whole flock will eventually have the same informations

To find a solution we just initialize particles randomly on the search space and move them based on OVERALL INFO and SELF INFO

↓

the balance is tuned.

With SWARM INTELLIGENCE tuning of the parameters can lead to huge differences ⇒ we use automatic parameter tuning which performs a local search in the parameter space.

GAME THEORY

We consider 2-players, MIN and MAX, alternating in a PERFECT KNOWLEDGE setting: each players have the same knowledge of the game.

A game match is a TREE where each leaf is a winning / losing condition. Each depth level corresponds to a player's turn, each player choose a path that minimize its choices to lose.

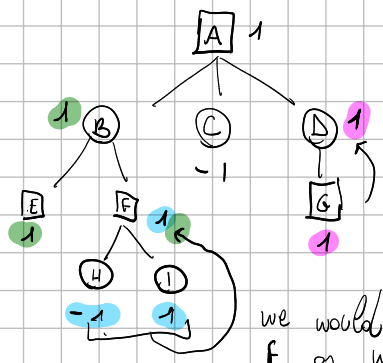
MIN-MAX ALGORITHM

Min max algorithm is based on finding at each MAX level the best move which will bring MAX to win.
MIN is assumed to always play the best move.

Each leaf contains +1 for MAX win and -1 MIN. Each parent node will contain the max value of children if its a MAX level or the minimum value of children.

MAX should therefore choose a move labeled as 1, taking to a path that will eventually bring him to the win.

eg.



we would like to end on F as we could go to I and win

MIN-MAX is optimal but very complex in time

$$O(b^m)$$

where usually
$b \approx 35$, $m \approx 100$

We want to stop the computation at a certain level:

- DEFINE A MAX DEPTH: Some moves however could be useless to explore as they would only give to the opponent a benefit.

- ALPHA-BETA CUT: If we know that along the same path we have a winning choice it doesn't make sense to explore other longer paths.

Through the use of ALPHA (for MAX) and BETA (for MIN) we keep track at each node the best solution found so far after that node.
We therefore avoid expanding a node that wouldn't bring an advantage.

Using ALPHA-BETA in fact reduces the search space because when we have 2+ choices we can directly discard one if, by choosing it, the opponent is in a better condition.

To evaluate a state we use an heuristic function which dictates how good (or bad) a move is.

By using ALPHA-BETA cuts we open the left-most subtree and then we start cutting subtrees depending on the first opened child of the other subtrees.
If a worst result than the actual ALPHA (or BETA) is immediately found than the tree is cutted otherwise if we found a better result we keep expanding.