Informed Search

① Search Paradigm.

- GTSP ⟷ GGSP
- GTSP
  - ↳ no dup det
  - ↳ stuck in cycles.
- GGSP
  - ↳ dup detection
  - ↳ might miss a better path through a visited node.

② Best First Search (BeFS)

- ↳ uses evaluation func^n ($f(n)$) — estimate cost from start to goal through some node n.
- ↳ when A* returns optimal path.
  - ↳ ① heuristic is consistent.
  - ↳ ② $\boxed{\forall n \in V \mid h(n) \le h^*(n) + (C_2^* - C^*)}$
  - where C_2 is the second best cost to reach goal.
  - ↳ It means we can overestimate a little.

③ Greedy best first search (GBeFS)

- ↳ $f(n) = h(n)$.
- ↳ choose next lowest $h(n)$.

④ GBeFS

- TC ⟶ $O(|V|) \equiv O(b^m)$
- SC ⟶ $O(|V|) \equiv O(b^m)$
- complete ⟶ yes: in GGSP only
- optimal ⟶ NO.

⑤ A* Search.

- ↳ $f(n) = g(n) + h(n)$
- ↳ will use min heap / PQ to store the fringe data.

if Δ inequality holds then heuristic is addmissible but not true for vice versa.

if heuristic is consistent.
- ↳ no need to re add state
- ↳ optimal path.

if $h(n)$ is inconsistent
- ↳ may need to re add
- ↳ sub optimal.

⑥ Properties of A*

① admisibility.

- ↳ heuristics that never overestimates.
- ↳ $\boxed{\forall n \in V \mid h(n) \le h^*(n)}$

where V = set of vertices.
$h(n)$ = estimated cost from n to goal
$h^*(n)$ = actual cost from n to goal.

⑧ ② Consistency.

- ↳ $f(n)$ is non decreasing / monotic
- ↳ cost can increase or stay same but can never decrease.

for every edge from u to v ⟶ $\boxed{h(u) \le C(u,a,v) + h(v)}$

where C is the cost to get from u to v by performing some action a.

- If $h(n)$ is addmissible.
  - ↳ run GTSP.
- If $h(n)$ is consistent.
  - ↳ run GTSP or GGSP.

Some other importances about A*

① if C^* is optimal cost then we will

- ⓐ always expand n if $f(n) \le C^*$
- ⓑ never expand n if $f(n) > C^*$

②

	$h(n) = h^*(n)$	$h(n) \approx h^*(n)$
TC	$O(d)$	$O(b^{d'})$
SC	$O(d)$	$O(b^{d'})$
Complete	yes	⟶ .
Optimal	yes^{a,b,c}	⟶

a: consistent / monotonic heuristics
b: no negative cycles.
c: no ∞ number of nodes with $f(n) < C^*$

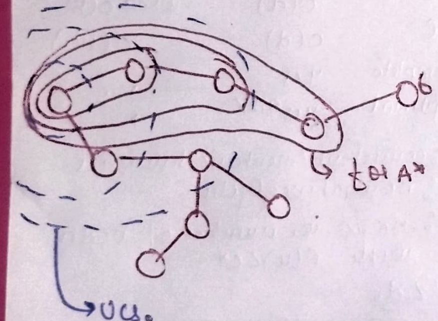
$d' < d$.

if $f(n) = \alpha g(n) + \beta \cdot h(n)$.

- $\alpha = 0$ ⟶ GBeFS
- $\beta = 0$ ⟶ UCS
- $\alpha = 1, \beta = 1$ ⟶ A*
- $\alpha = 1, \beta = W$ ⟶ weighted A*

↳ IDA*.

* Search Contours.



* Some other variants of A*.

① Bounded suboptimal.
↳ we search for $f(u) \le w \cdot C^*$
$w \ge 1$, if $w = 1.2$, our sol^n is 20% worse than optimal.

② Bounded cost search.
↳ we increase w till $w \le B$.
↳ like IDA*.

↳ Beam Search.

① keep best k nodes of $f(u)$ value in fringe.

② keep nodes with $f(u) = f^*(u) - \Delta$ in fringe.
↳ does not have fixed fringe len. but keeps good nodes around n.

↳ Depth first branch & Bound (DFS B&B)

↳ Branch policy → lowest edge first.
↳ keeps
global max
local max.
↳ at start global max = ∞.

Algo
global max = ∞.
for node in fringe:
  local max = 0
  for vertex in path:
    local max += path cost
    if (local max > global max)
      prune
      break.
  if (local max < global max)
    global = local.

DFS B&B may search nodes with $F(u) > C^*$.
↳ optimal → yes
↳ complete → yes?
as depth is finite.

* heuristic func^n.

* Effective branching factor → average branching per node.
* Effective depth → depth at which sol^n may be found.

$\boxed{N+1 = 1 + b^* + (b^*)^2 + \dots (b^*)^d}$ → N is the no. of nodes needed to be expanded in order to reach sol^n at depth d.

if K_u is depth of sol^n then TC of A* becomes $O(b^{d-K_u})$
∵ A* prunes not essential nodes.

closer a heuristic is to the actual cost its performance will be greater.

* Pattern databases.
↳ divide the problem into multiple subproblems (may or may not overlap).
↳ solve them by disregarding other elements.
↳ sum all of their cost.

* Landmark points.

$\boxed{h_L(u) = \min C^*(n,L) + C^*(L, goal).}$

$\boxed{h_{DH}(n) = \max |C^*(n,L) - C^*(goal, L)|}$ ↳ Differential heuristic

Algo.	neg edge	neg cycle.
① GBeFS	Unaffected	Unaffected
② A*	may get affected. ↳ consistency is broken	↳ completely fails.
③ WA*	Same	Same
④ IDA*	↳ may prune the optimal path.	↳ fails.
⑤ DFS B&B	↳ may fail ↳ may prune the goal path	↳ completely fails.

Local Search

↳ Searches in the goal state space and finds a good enough or best state.

goal Satisfaction ⟷ optimization (can be converted)

↳ both are in same complexity class.

* Random walk or Random restart.
↳ both are asymptotically complete i.e if you give them enough time you will get a solⁿ.

* greedy hill climbing
↳ never backtracks
↳ only one and any neighbour.
↳ cannot escape local optima alone.
↳ highly dependent on initial state.

Draw backs
↳ local maxima
↳ plateu
↳ Diagonal ridges.

8 queens solⁿ rate

max | stuck

14% in 4 steps | 86% in 3 steps
94% in 21 steps | 6% loop
↳ with 100 sideways step.

Local Search.

* Calculate steps and expected iterations.

P = probability of success

1/p = Expected no. of iterations.

* Expected no. of steps =

$$\left[\frac{1}{p} \times \text{steps to reach max}\right] + \left[\frac{1-p}{p} \times \text{steps to get stuck in local minima}\right].$$

ex: p = 14% in 4 steps
$\bar{p}$ = 86% in 3 steps.

Expected no. of iterations
$= \frac{0.14}{4} \approx 7.$

expected no. of steps. =
$(7 \times 4) + (6 \times 3)$
$= 46$ steps.

* Problems with Plateu.
↳ if shoulders are fully connected then search won't stop.
↳ hence we introduce tabu search which keeps K previous nodes.

* Tabu search.
if K = 1
↳ It becomes greedy hill climb
if K = no. of nodes / or large
↳ It becomes systematic search.

if size of shoulder is greater than size of tabu list we again have issue of infinite search.

Enforced hill climbing.
↳ greedy hill climbing or any local search till local minima
↳ systematic search to get out.

Stochastic beam search is not equivalent to parallel running greedy hill climbing.

* Simulated annealing.
↳ Its tunable parameters
① starting prob.
② decay rate at which parameters will decay.

↳ for state x, select random neighbor y. if δ > 0 → move to y
else → move to y with prob $e^{\Delta E/T}$.

* Local beam search.
↳ keep track of K state
↳ run algorithm on all state simultaneously
↳ but select K best successor from every successor generated.
↳ may result in selection of variable from 1 state only.
↳ solⁿ stochastic beam search.
↳ select K neighbour randomly biased towards good one.

* GA.
↳ Combine to state to generate a child state.
↳ It has a lot of tunable parameters.
↳ It uses a pratice called elitism.
↳ Stages
① Random selection.
② crossover.
↳ results in jumping out of local minima.
③ Mutation.
↳ helps make small jump.
↳ This part is like local search.

for jumping, i.e selecting random State from a local ~~minima~~ max., our jump should be big enough to get out of local minima but ~~not~~ ~~big~~ small enough to not jump out of global. max.

gradient descent.

for any eqn

$$y = w_1 x_1 + w_2 x_2 + \dots w_n x_n.$$

change in i should be.

$$\boxed{x_i = x_i - \frac{dy}{dx_i}.}$$

$\forall n \in \{1, 2 \dots n\}.$

Adversarial Search And Games.

Adversarial Search.

↳ Search by predicting move of the opponent.

↳ not similar to ~~[illegible]~~ search as here we can't say for sure what opponent does.

* Configuration of A.S
  1. States
  2. Initial state
  3. Successor funcⁿ
  4. Terminal test
  5. Utility funcⁿ.

(#) In adversarial search.

max:
↳ tries to maximize its utility funnc

min:
↳ minimize our utility funcⁿ

* Minimax algorithm

$$= \begin{cases} Utility(S, max) \rightarrow leaf. \\ max(minimax(S,a)) \\ min(minimax(S,a)) \end{cases}$$

min move — max move

~~*Opti*~~

α-β ~~pos~~ pruning

1. Starting by (-∞, ∞)
2. two rules.

(a) Pass α, β as it is to child.

(b) To parent pass ~~the~~ the node value such that

```
if Parent is max:
    ~~Parent state~~
    if (Parent.α > node.value)
        Parent.α = node.value
    }
else {
    if (Parent.β < node.value)
        Parent.β = node.value
}
```

if (α >= β) Prune.

* Move ordering.

↳ max player
  ↳ best: ~~as~~ descending order
  ↳ worst: ascending order.

↳ min player
  ↳ best: ascending order
  ↳ worst: descending order.

α-β pruning (normal)

recurrence	T.C
$T(m) = b.T(m-1) + c$	$O(b^m)$

ideal pruning.

$T(m) = T(m-1) + (b-1)T(m-2)$	$O(b^{m/2})$

(#) for a best ordering we can search twice as deep.

↳ we can use iterative deepening to order the move.

↳ Killer moves : known to be best.

↳ Transposition : different permutation of same seq. that end up on same state we resolve this by transposition table.

Type A strategy

↳ Consider all possible move to a depth a

Type B strategy

↳ Ignore moves that looks bad and search nodes.

* Cutoff Search.

↳ Instead of going to the leaf cutoff at a certain depth.

↳ now we define an Eval funcⁿ to calculate the value of nodes.

$$\boxed{Eval(s) = w_1 f_1(s) + w_2 f_2(s) \ldots \ldots w_n f_n(s).}$$

↳ weighted linear funcⁿ or basis funcⁿ.

(#)

$$\boxed{Eval(S) \leq Utility(win)}$$

where S is state at cutoff and win is ~~[illegible]~~ leaf node.

~~Expected Cost~~

Quiescence Search

↳ Positions which would swing the evaluations wildly (such as capturing queen).

↳ we search till we find such nodes.

Horizon effect

↳ not being able to see a obvious bad move just after the cut off.

↳ or not being able to see a good ~~[illegible]~~ move.

Later move reduction

- We assume that the moves are ordered well and then we only search for smaller amounts for later moves.

* Additional Refinements

1. probabilistic CUT.
   - Cut branches based on shallow search.
2. opening and endgame databases.

Types of games

	Deterministic	Chance
Perfect Info		
Imperfect info		

Games of chance.

- Calculation of chance node.

$= Ex(x_i p_i)$

where x_i is value of i^{th} child and p_i is ~~value~~ probability of that i^{th} child.

Constraints on.

Constraints Satisfaction

* CSP
↳ It has 3 Components
① X set of Variables
② D set of domains.
③ Y set of constraints.

* Types of Constraints
① precedence Constraints.
↳ before T_1 finish T_2
② disjunctive Constraints.
↳ If two entities utilize a single resource then they must not overlap
③ Linear Constraints.
↳ Each constraints appear in linear form.
④ Non linear, Undecidable.

* Types of Variables.
① Unary constraint.
↳ restrict value of a single variable.
really good for domain reduction
② Binary Constraint
↳ constraints that involve exactly 2 variables
↳ The variables in itself can be Unary.

③ Higher order.
↳ any thing with more than 2 variables.

* Global Constraint.
↳ Constraint over a set of variables. for those set of variables it is a global constraint.

* Node Consistency
↳ for every variable in the Problem, Every value in its domain satisfies the variables Unary constraints.

* ARC consistency
↳ for every variable X and every value α in its domain
↳ there exists a value β in Y domain such that the binary constraint b/w x and y satisfies.
where Y are all of X's neighbours

* Path Consistency
↳ If for every variable pair (X,Y) and value (a,b) that satisfies constraints b/w X and Y
↳ There exists c in third variable Z such that (XZ), (YZ) satisfies

In constraint graphs.
↳ nodes are variable
↳ edges are constraints.

if D has a size d then we have $O(d^n)$ complete assignment

Standard Search formulation of CSP.
↳ at length l we have $(n-l)d$ leaves
↳ it at max we have $\boxed{n! d^n \text{ leaves.}}$
↳ to solve this at every level we assign only 1 variable.
↳ $\boxed{n = d^n \text{ leaves.}}$

* Improving Backtracking efficiency
① which variable should be assigned next.
↳ ① MRV (minimum remaining value).
↳ choose the variable with minimum possible value.
② Degree heuristic
↳ select variable with most constraints on.

② In what order the values should be tried.
↳ LCV (least Constraining Values).
↳ the one which rules out fewest value in the remaining variables.

③ Can we detect inevitable failures early.
① forward checking
↳ check b/w assigned and non assigned.
↳ after assigning a value check non-assigned nodes that what values are not possible.
② ARC consistency
↳ check b/w ~~multiple~~ all non-assigned.
↳ remove the value which causes issue and then recheck.

X → Y is consistent iff
$\forall x \in X \ \exists y \in Y$ where
y satisfies X.

↳ $\boxed{O(n^2 d^3)}$

(#) for k consistency our space size will be $O(d^k)$.

(#) if $k=n$ it becomes for inferential algo and $k=0$ it becomes a search algo.

(4) <u>problem structure</u>

↳ divide into subproblem if k connected components are present.

↳ $n/c \, d^c$ complexity

where c = no. of variables in each sub problem.

(2) Tree structure CSP's

(#) can only be done by inference no search req.

↳ $O(nd^2)$

Steps

(1) from n to 2

↳ Remove inconsistency (Parent(X_j), X_j)

(2) from 1 to n.

↳ assign X_j

(3) nearly tree structured CSP.

↳ <u>cutset.</u>

↳ if we have a cutset of size c

$\boxed{O(d^c \cdot (n-c)d^2)}$.