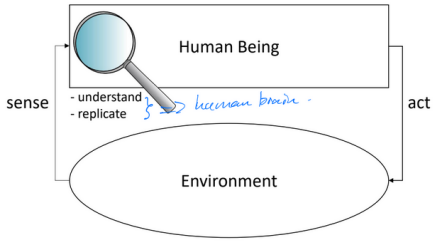
"Cognitive Science" -> understand and replicate
Engineering Approach: agent = software system not Human Being

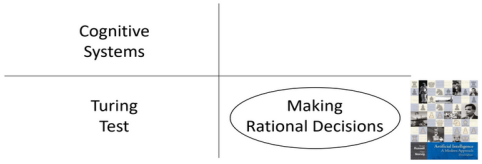


Turing Test =>standard of AI=> Not good to drive AI research but good for test AI
problem: 1. we cannot talk about very deep topic; 2,non-nativespeaker, cannot talk about something that is not exist.
Problem of AI: may quickly become racist as learn from incorrect content
Incremental progress might fool some judges but might eventually not result in truly intelligent systems.
Winograd Schema Challenge=> rule: make statement and then ask question => A because B, what is B requires the reasoning about the properties of different object in these sentences but nothing about the structure of these sentence => common sense and logics

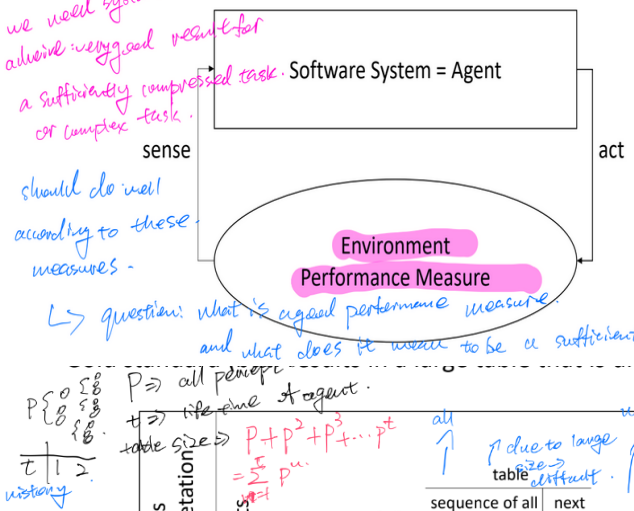
A different view: What is AI?

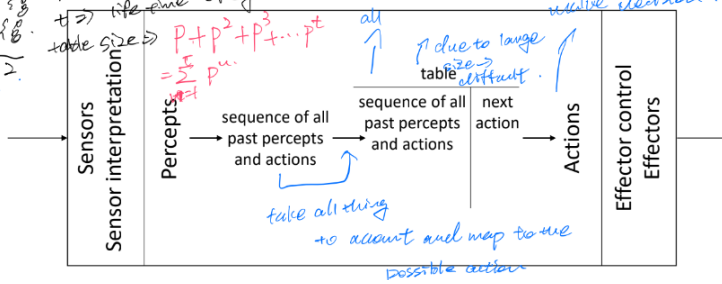
Systems that think like humans	Systems that think rationally
Systems that act like humans	Systems that act rationally

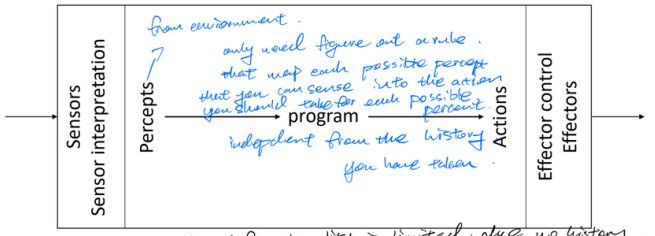
A different view: What is AI?

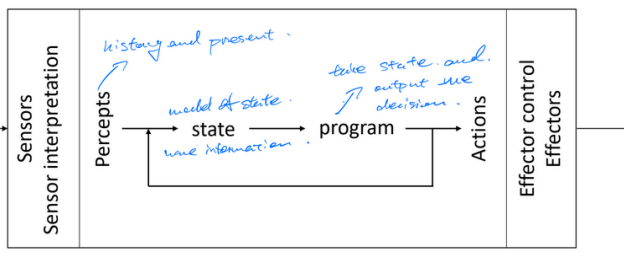


"Making Rational Decisions"









Architecture for planning agent
Gold standard, but result in a large table that is difficult to change
Reflex(simpler) agent (" reactive planning"): often good for video games as all possible info is already in the environment

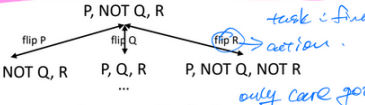
Usually not need remember the sequence of past percept and action to perform well according to its performance object (just need some part of the history as other parts are less important)

State: the information that an agent needs to have about the past and present to pick actions in the future to perform well (information that is required for decisions)

Typically interested in menial states => simple but would have important information

Goal states: the optimal state that we want to achieve, states are important for make decision and measure performance
State Space:

- S ≡ (P OR Q) AND (NOT P OR NOT R) AND (P OR NOT Q OR R)
- start state = assignment of random truth values to all propositional symbols
- Costs do not matter since we are not interested in finding a minimum-cost path.
- There is more than one goal state, e.g. P, Q, NOT R and P, NOT Q, NOT R.



• Graph	• State space
• Vertex	• State
• Edge	• Action = operator = successor function succ(s,s') ε States
• Edge cost	• Action cost = operator cost
• Start vertex	• Start state
• Goal vertex	• Goal state or goal test goal(s) ε {true, false}
• Solution is a (minimum-cost) path from the start vertex to any goal vertex	• Solution is a (minimum-cost) action sequence = operator sequence from the start state to any goal state

Problem: 1. Single-agent path-finding problems 2. Two-player games 3. Constraint satisfaction problems
State space=> special case for problem space
Single-agent path-finding problems:in each case, we have a single problem-solver making the decisions, and the task is to find a sequence of primitive steps that take us from the initial location to the goal location. (eg: traveling salesman problem. Rubik's cube 魔方) **will have a map for the problem.**
Two-Player Games
one must consider the moves of an opponent,and the ultimate goal is a strategy that will guarantee a win whenever possible. Eg. chess
Constraint-Satisfaction Problems
We also have a single-agent making all the decisions, but here we are not concerned with the sequence of steps required to reach the solution, but simply the solution itself.
The task is to identify a state of the problem, such that all the constraints of the problem are satisfied. **Eg. Eight Queens Problem**
Problem Spaces
A problem space also consists o a set of states of a problem and a set of **operators** (a function that takes a state and maps it to another state [action]) that change the state.
Not all operators are applicable to all states. The conditions that must be true in order for an operator to be legally applied to a state are known as the **preconditions** of the operator. Eg: in chess, not all place is available for moving under rules

All 4 combinations are possible: [single/set of goal state(s)] x [explicit\implicit]
A problem instance: consists of a problem space, an initial state,and a set of goal states.
For Constraint Satisfaction Problems, the goal will always be represented implicitly, since an explicit description is the solution itself.

How to find the problem state? . Easy example – maps not so easy —--theorem proving how to represent the problem can influence the efficiency of solving the problem.

A smaller representation-> fewer states-> better than larger one (which means **easier to solve)**

How to represent problems

- *Example*, in the 8-Queens problem, when every state is an assignment of the 8 queens on the board:
  - The number of possibilities with all 8 queens on the board is 64 choose 8, which is **over 4 billion.** States .
  - The solution of the problem prohibits more than one queen per row, so we may assign each queen to a separate row, now we'll have 8^8 > **16 million** possibilities.
  - Same goes for not allowing 2 queens in the same column either, this reduces the space to 8!, which is only **40,320** possibilities.

problem space.

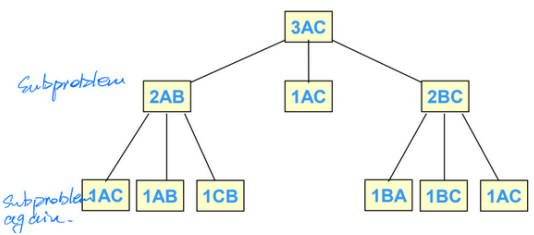
Problem-Space Graphs => a mathematical abstraction often used to represent a problem space:
The **states** => vertices => situation of the problem
The **operator** => edges (undirected or directed) => action in the world weights are the cost of the operator
If we want the goal test (no need care edge cost) we can only have cost of each state

If have same states in different phase of the graham then we may have two search nodes that point to the same state, which create two different states on the graph
In most problem space there is more than one path between a pair of vertices => multi goal state
Duplicate detection requires saving all the previously generated states and comparing new states with the old ones. Most algorithms don't do detection due to memory saving and simplicity. (just seen those nodes as two different unique nodes)

Branching factor of a state b(n): the number of children it has, not counting its parent if the operator is reversible.
Branching factor of a problem space b: is the average number of children of the vertices in the space.
Solution depth in a single-agent problem d: the length of the shortest path from the initial vertex to a goal vertex. A function of the particular problem instance.

Types of problem space: state space(OR graphs); Problem Reduction Space (AND graphs); Games (AND/Or graphs)

State Space:
Forward search: the root of the problem space represents the initial state, and the search proceeds forward to a goal state.
This can be considered as an OR graph because the solution picks one branch at each vertex. Solution can be only apply on the action, so that solution can be seen as a path



Problem Reduction Space: AND graph
The vertices represent problems to be solved or goals to be achieved, and the edges represent the decomposition of the problem into subproblems. Eg: Towers of Hanoi problem.

AND/OR Graph

An AND graph consists entirely of AND vertices, and in order to solve a problem represented by it, you need to solve the problems represented by all of his children

An OR graph consists entirely of OR vertices, and in order to solve the problem represented by it, you only need to solve the problem represented by one of his children

An **AND/OR graph** consists of both AND vertices and OR vertices. a problem where the effect of an action cannot be predicted in advance, as in an interaction with the physical world. Eg, Two player games => no full control of the world (opponent in game)

Solution subgraph for **AND/OR graphs**:

A solution to an AND/OR graph is a subgraph with the following properties: It contains the root vertex. For every OR vertex included in the solution sub graph,one child is included. For every AND vertex included in the solution subgraph, all the children are included. Every terminal vertex in the solution subgraph is a solved vertex.

Path-finding:optimal solution is lowest cost

CSP: if there is a cost function associated with a state of the problem, an optimal solution would again be one of lowest cost, the lowest-cost goal state.

2-player game: optimal solution is the best possible move that can be made in a given situation. If the solution is considered a complete strategy subgraph, then an optimal solution might be one that forces a win in the fewest number of moves in the worst case.

The most common source of AND/OR graphs is 2-player perfect-information games.

Search Algorithms

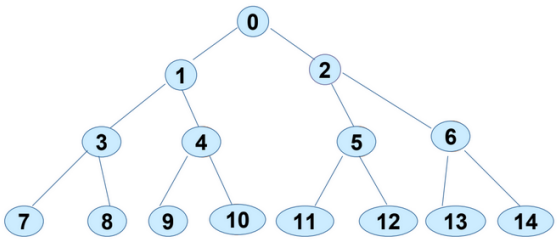
Brute-force: uninformed search => no knowledge guide the decision

If there is a node that has no information that is outside of the tree or that part has not been constructed or discovered, then it is uniformed.

Heuristic: informed search => has knowledge for each decision

4 primary performance measures: completeness, Time complexity. Space complexity, optimally

Breadth-First Search: If a goal exists in the tree BFS will find the shortest path to a goal.



Pruning rule: do not expand a node if a node labeled with the same state has already been expanded. Thus, we can say that states get expanded rather than nodes.

Optional termination rule: terminate once a node labeled with a goal state has been generated.

Two ways to report the actual solution path:

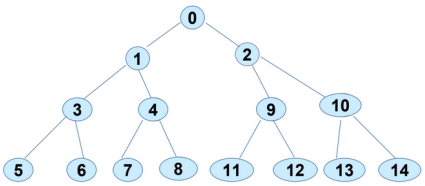
• Store with each node the sequence of moves made to reach that node.

• Store with each node a pointer back to his parent - more memory efficient.

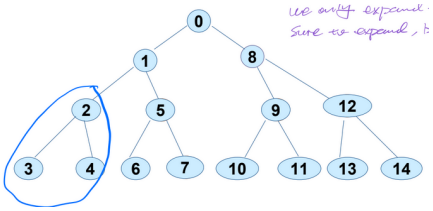
N(b,d) - total number of nodes generated.

Space Complexity=Time Complexity= O(bSpace Complexity=Time Complexity= O(bd)Space Complexity=Time Complexity= O(b^d))

Depth-First Search: **not** implemented recursively



Depth-First Search: **not** implemented recursively



DFS may goes an infinite loop as their will be a path which explored a node which is generated before => A->B->C and non is goal

DFS(memory inefficient)

Pruning rule: break cycles, for example, do not expand a node if a node labeled with the same state has already been expanded. Thus, we can say that states get expanded rather than nodes. **Duplicate detection as before**

Optional termination rule: terminate once a node labeled with a goal state has been generated.

DFS (memory efficient) delete unnecessary subtree from search tree **The new pruning rule is very weak.**

Pruning rule: break cycles, for example, do not expand a node if a node labeled with the same state exists from the root node to the node in question.

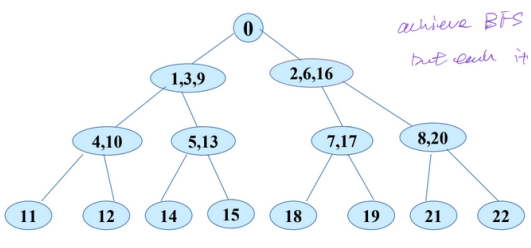
• **Optional termination rule:** terminate once a node labeled with a goal state has been generated.

Iterative Deepening Search: limited depth and append the depth by one each time

DFID first performs a DFS to depth one. Then starts over executing DFS to depth two. Continue to run DFS to successively greater depth until a solution is found.

Implement a breadth-first search with a series of depth-first searches with increasing depth limits (that is, depth-first searches that assume that nodes whose depths are at least the depth limit have no children).

Iterative deepening incurs a time overhead over breadth-first search since it expands nodes multiple times. But the overhead is small and thus a small price to pay for the small space complexity.



If the branching factor is two, then about half of the expanded nodes are expanded for the first time during each depth-first search. This percentage is even larger for larger branching factors.
Since DFID uses O(d) space, it's asymptotically optimal in space.
Backward Search: The root node represent the goal state, and we could search backward until we reach the initial state.

Bidirectional Search: Simultaneously search forward from the initial state and backward from the goal state, until the two search frontiers meet at a common state.
Bidirectional search guarantees finding a shortest path from the initialstate to the goal state, if one exist.
When the backward search has proceeded to depth d-k, its frontier will contain all states at depth d-k from the goal state.
The total number of nodes generated is O(2bd/2) = O(bd/2).
The simplest implementation of bidirectional is to use one search in BFS, and the search in other direction can be DFS such as DFID.
Bidirectional search is space bound Bidirectional search is much more time efficient than unidirectional search..
Best-First Search eg A-star breadth-first search, no DFS
Open list and closed list(node expanded), cost , length
The Open list is maintained as a priority queue.
Uniform-Cost Search:
Let g(n) be the sum of the edge costs from root to node n. If g(n) is our overall cost function, then the best-first search becomes Uniform- Cost Search,
Pruning rule: do not expand a node if a node labeled with the same state has already been expanded. Thus, we can say that states get expanded rather than nodes.
We consider Uniform-Cost Search to be brute force search, because it doesn't use a heuristic function. (all info is from the current tree)

Algorithm 2: Best-first search algorithm

```
Input: Source vertex s
OPEN.insert(s)
while OPEN ≠ ∅ do
    u = OPEN.extract_min()
    foreach vertex v ∈ Adj(u) do
        g(v) = g(u) + w(u, v)
        v' = check_for_duplicates(v)
        OPEN.insert(v')
        CLOSED.insert(u)
```

Algorithm 1: Dijkstra's algorithm

```
Input: Graph G = (V, E)
(∀x ≠ s)dist[x] = +∞   //Initialize dist[]
dist[s] = 0
S = ∅
Q = V   // Keyed by dist[].
while Q ≠ ∅ do
    u = extract_min(Q)
    S = S ∪ {u}
    foreach vertex v ∈ Adj(u) do
        dist[v] = min(dist[v], dist[u] + w(u, v))
        //"Relax" operation.
```

In a graph where all edges have a minimum positive cost, and in which a finite path exists to a goal node, uniform-cost search will return a lowest-cost path to a goal node.
UCS is O($b^{c/e}$)
Dijkstra's algorithm is the same as uniform search
(1) Applicability:
• DA: Explicit graphs only
• UCS: Explicit and Implicit graphs

• (2) Memory needs:
• DA: |Q|+|S|=|V| - the entire graph is stored
• UCS: Only Open and Closed

Informed Search
The efficiency of an uninformed (brute-force) search can be greatly improved by the use of a heuristic static evaluation function, or **heuristic function.**
leading the algorithm toward a goal state
pruning off branches that don't lie on any optimal solution path.
Properties:
• it is relatively cheap to compute
• it is a relatively accurate estimator of the cost to reach a goal state. Usually a heuristic is (roughly) "good" if 1⁄2 opt(s)<h(s)<opt(s)
Admissibility - the heuristic function is always a lower bound on actual solution cost. (always under-estimating)
Manhattan Distance
Heuristic function: Manhattan distance - number of horizontal and vertical grid units where each tile is displaced from its goal position
Traveling Salesman Problem
Heuristic function:A cost of minimum spanning tree (MST) of the cities.
Pure Heuristic (Greedy Best-First) Search => not optimal
Pick an unexpanded fringe node n with the smallest f(n) = h(s(n))
Optional pruning rule: do not expand a node if a node labeled with the same state has already been expanded. Thus, we can say that states get expanded rather than nodes.

Different cost combinations of g and h
- **f(n)=level(n)** Breadth-First Search.
- **f(n)=g(n)** Dijkstra's algorithms. Uniform-cost Search
- **f(n)=h(s(n))** Pure Heuristic Search (PHS).
- **f(n)=g(n)+h(s(n))** The A* algorithm (1968).

Optional termination rule: terminate once a node labeled with a goal state has been generated.
A* Pick an unexpanded fringe node n with the smallest f(n) = g(n) + h(s(n)), g(n)=> cost of reach n, h(s(n)) => cost of reach goal from nAdmissible condition: the estimated cost to goal always underestimates the real cost h(s(n)) <= gd(s(n))
•when h(s(n)) is admissible, so is f(n):f(n) <= gd*
In general, A* is guaranteed to return optimal solutions only if the heuristic is admissible.
Admissible H-Values: lowest cost by ignoring the constraint
Consistent H-Values: if and only if they satisfy the triangle inequality (c(s,s') is the action cost of moving from s to s'): h(s) = 0 for all goal states s, and
0 <=h(s) <=c(s,s') + h(s') for all non-goal states s and their successor states s'.
Problem Relaxation:Obtain a new planning problem by relaxing constraints of the actions (e.g. by deleting preconditions of operator schemata), which can add states and actions to the state space.

To verify that h-values are consistent, either prove that the triangle inequality holds or show that they can result from a problem relaxation.
To create consistent h-values, create admissible h-values and verify that they are consistent.

Dominating H-Values:H-values h(s) dominate h-values h'(s) if and only if, for all states s, h(s) ш h'(s).
when A* uses an admissible heuristic h(n), it returns an optimal solution when the goal node is chosen for expansion
Tie Breaking: always break ties among nodes with the same f(n) value in favor of nodes with the smallest h(s(n)) value (or the largest g(n) value).
Conditions for Node Expansion: f(n)<=c
The main drawback of A* is its space complexity.

MAPF
Vertex collision and **edge collision**

Suboptimal MAPF algorithms: MAPF can be solved in polynomial time on undirected grids without makespan or flowtime optimality
Optimal MAPF algorithms: MAPF is NP-hard to solve optimally for makespan or flowtime minimization

Prioritized Planning (= sequential search: plan for one agent after another in space (= cell)-time space in a given order): efficient but suboptimal (and even incomplete) MAPF solver

CBS:Optimal (or bounded-suboptimal) MAPF solver that plans for each agent independently, if possible
Solution quality: For a given node N in the search tree, let CV(N) be the set of all plans that are: (1) consistent with the set of constraints of N and (2) are also valid (i.e., without conflicts).
We say that node N permits a plan p if and only if p∈CV(N).
For every cost C, there is a finite number of nodes with cost C.

Suboptimal Searches
Suboptimal search: Just find any solution fast: Speedy vs. Greedy.
• **Bounded suboptimal search**: Given a bound W we want the solution to
be at most W x C* This is called w-admissible.
• **Anytime search**: improve the quality as time passes
• **Bounded cost search**: Give B, find a solution below B
• **PAC search:** given e and d, find a solution that will be of quality e with probability b
W-admissible: Given a bound W we want the solution to be at most W x C*
Weighted A*::f(n) = g(n) + Wh(n)
Small W – more time better solution
Large W – less time worse solution
Given W, WA* will find a solution which is no more than W times the optimal solution

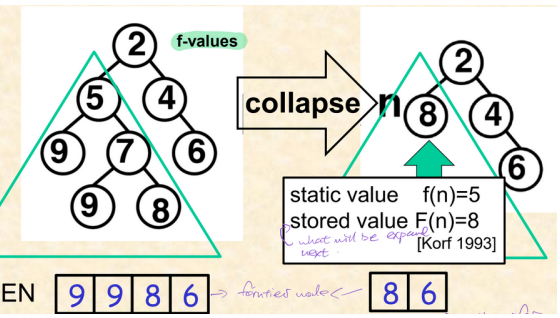
Algorithm 1: Focal Search: main procedure

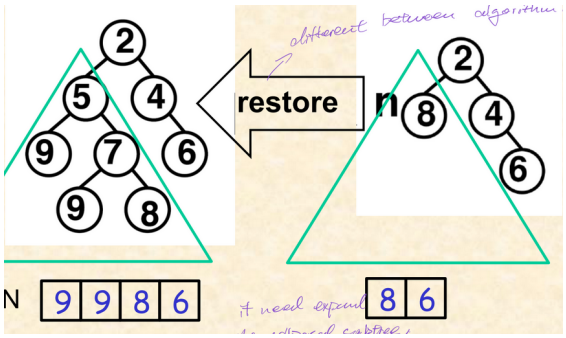
```
1  focal-search(start state S)
2      OPEN ← {S};
3      FOCAL ← {S};
4      while FOCAL ≠ ∅ do
5          best ← ChooseNode(FOCAL)
6          Remove best from FOCAL and OPEN
7          if best is a goal then return best;
8          if f_min increased then FixFocal();
9          for n ∈ neighbors(best) do
10             Add n to OPEN
11             if f(n) ≤ B × f_min then add n to FOCAL;
12         end
13     end
14 end
```

Focal search
Sort OPEN according to the f-value
• Keep a FOCAL list of nodes from OPEN
• Each node n in FOCAL has
• Fmin ч f(n) ч W * Fmin
• Then, expand a node from Focal via a secondary function.
Any node in FOCAL is guaranteed to be W-admissible
Potential Search: Choose to always expand the node n in open with maximal: u(n) = (c-g(n))/h(n)
Iterative weighting A*: decrease w in each iteration
Iterative Deepening A*: set a cost, for each iteration if f=>cost of the iteration search is larger than cost, terminate search, increase cost threshold and do next iteration
If a solution of finite cost exists, IDA* will find and return one.IDA* will return an optimal path to a goal.
Time Complexity
The last iteration will expand all nodes connected to the root whose cost is less than or equal to c.
In the worst case (when the heuristic function is not consistent) IDA* expands the same set of nodes as A* does.
IDA* is much easier to implement then A* because it's a DFS algorithm and no open and closed lists have to be kept.
limitations of IDA*:When all the node costs are different:
• IDA* will develop a different iteration for each node and in each iteration only one new node will be expanded.
• On such a tree the time complexity of A* will be O(bd) but for IDA* O(b2d).
• If the asymptotic complexity of A* is O(N) - IDA*'s complexity can get in the worst case to O(N2).
The problem space for IDA* must be a tree because :
• if a certain node can be reached via multiple paths it will be represented by more than 1 node in the search tree.
• A* can avoid the duplicate nodes by storing them in the memory but IDA* is a DFS (no memory) and thus it can not detect most of the duplicates.
• This can increase the time complexity of IDA* compared to A*.
• Thus, if there are many short cycles in the graph and there is no memory problem - choose A*.
IDA* isn't so effective for all the problems. Eg TSP
Depth-First Branch-and-Bound is often used when the optimal solution is required.

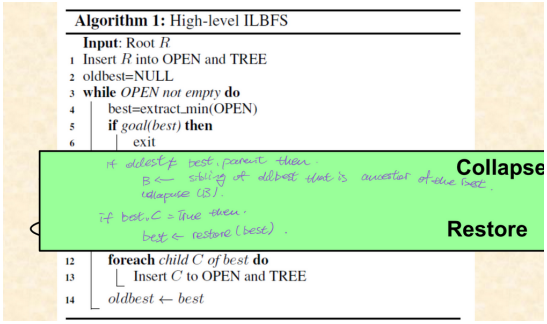
The asymptotic complexity is O(bd),
DFBnB starts with an upper bound cost threshold and decreases through the search.

Recursive Best-First Search
RBFS only keeps the current search path and the sibling nodes along the path
RBFS is a linear-space algorithm that expands nodes in best-first order even with a non-monotonic cost function and generates fewer nodes than iterative deepening with a monotonic cost function
RBFS generate fewer nodes than ID on average. Because RBFS only backtracks to their common ancestor instead of directly to the root as IDA*.

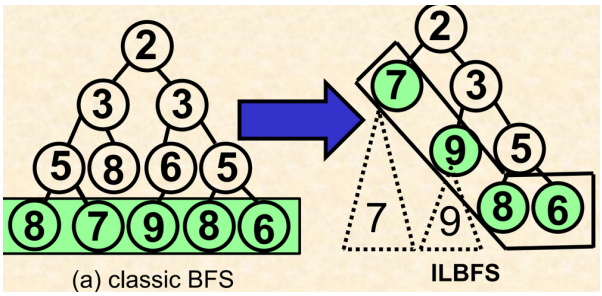




Iterative linear best-first search

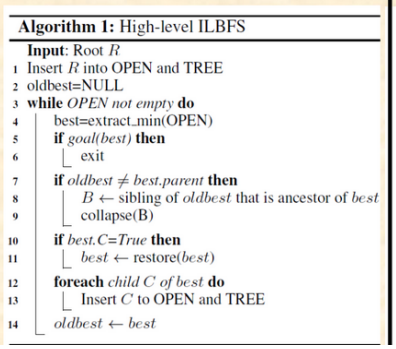


Store only the branch of the best node and its siblings

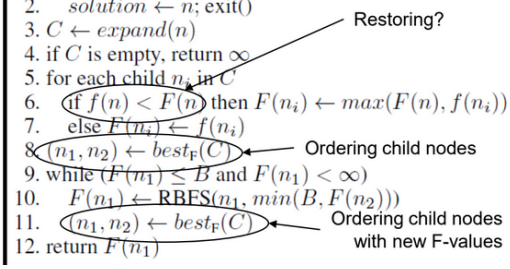


Principal branch invariant:
Initially valid for the root, Two cases for the expansion cycle.

Iterative variant - ILBFS



Recursive variant - RBFS



Simplified Memory Bounded A*
If child node larger than the bound collapse the child node subtree
It is complete, provided the available memory is sufficient to store the shallowest solution path.
• It is optimal, if enough memory is available to store the shallowest optimal solution path. Otherwise, it returns the best solution (if any) that can be reached with the available memory.
• Can keep switching back and forth between a set of candidate solution paths, only a few of which can fit in memory (thrashing)
• Memory limitations can make a problem intractable wrt time
• With enough memory for the entire tree, same as A*

Predicting the Time Complexity of A*
The impact of this assumption is that once we diverge from the optimal path from the root to the goal, the only way to reach the goal is to backtrack until we rejoin the single optimal path.
Constant Absolute Error:

Assume that the path from the start node s to node n diverges from the path to the goal at node m. Let d be the distance from s to g , x to be the distance from s to m, and y the distance from from m to n. Thus: y<=k/2
A much more realistic assumption for measurements of heuristic functions, is constant relative error. We assume that the absolute error is a bounded percentage of the quantity being estimated.

Limitation

There are several limitations of this model:

The abstract model makes unrealistic assumptions. Most real problem spaces, such as Rubik's cube or the sliding tile puzzles, are graphs with cycles as opposed to trees. In such problem spaces we can reach the goal from any other state without backtracking along the path to the given state.

In order to determine the accuracy of the heuristic on even a single state, we need to determine the optimal solution cost to a goal from the state, which requires a great deal of computation and impractical for the size of problems we are trying to solve.

We characterize a heuristic by the distribution of heuristic values over the states in the problem space.

Nodes Expanded as a Function of Depth

We will count the number of node expansions, which is the number of fertile nodes in the graph. The number of nodes generated is simply b times the number expanded.

Due to consistency of the heuristic function, all the possible parents of fertile nodes are themselves fertile. Thus, the number of nodes with each heuristic value to the left of the diagonal line is the same as it would be in a brute force search to the same dep**th.**

Conditional Distribution Prediction

Instead of p(v) – static distribution, we store p(v,vp) the probability that a value of a node with state s is h(s)=v given that its parent node with state p value was h(p)=vp.

Local Search

involve finding a grouping, ordering, or assignment of a discrete set of objects which satisfies certain constraints

eg:finding shortest/cheapest round trips (TSP)

start from initial position

• iteratively move from current position to neighboring position

• use evaluation function for guidance

Two main classes:

• local search on partial solutions

• local search on complete solutions

Limited Discrepancy Search • At each node, the heuristic prefers one of the children. • A discrepancy is when you go against the heuristic. • Perform DFS from the root node with k discrepancies • Start with k=0 • Then increasing k by 1 • Stop at anytime.

Limited Discrepancy Search

anytime algorithm

• Solutions are ordered according to heuristics

Iterative Improvement for SAT

initialization: randomly chosen, complete truth assignment • neighborhood: variable assignments are neighbors iff they differ in truth value of one variable • neighborhood size: O(n) where n = number of variables • evaluation function: number of clauses unsatisfied under given Assignment

Stochastic Local Search

• randomize initialization step • randomize search steps such that suboptimal/worsening steps are Allowed • improved performance & robustness • typically, degree of randomization controlled by noise parameter

Simulated annealing

Combinatorial search technique inspired by the physical process of annealing [Kirkpatrick et al. 1983, Cerny 1985]

= Hill Climbing with going downhill from time to time

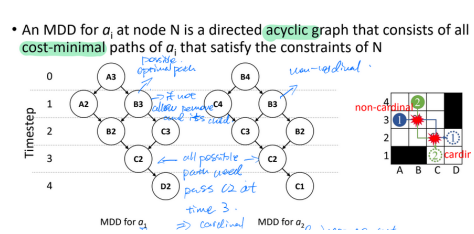
• Annealing: the process of gradually cooling a liquid until it freezes If the temperature is lowered sufficiently slowly, the material attains a lowest-energy (= perfect ordered) configuration.

is historically important • is easy to implement • has interesting theoretical properties (convergence), but these are of very limited practical relevance • achieves good performance often at the cost of substantial run-times

Tabu Search

Combinatorial search technique which heavily relies on the use of an explicit memory of the search process [Glover 1989, 1990] to guide search process • memory typically contains only specific attributes of previously seen Solutions • simple tabu search strategies exploit only short term memory • more complex tabu search strategies exploit long term memory

Improved MAPF Algorithms



Detecting Dependency

Two agents are dependent iff every pair of their optimal paths has at least one conflict

A pair of dependent agents is an admissible h-value of 1

Two agents that have cardinal conflicts are dependent

The weight for a pair of agents is the difference between the minimum sum of the costs of their conflict-free paths and the sum of their individual optimal path costs

The weight is an admissible h-value for the pair of agents

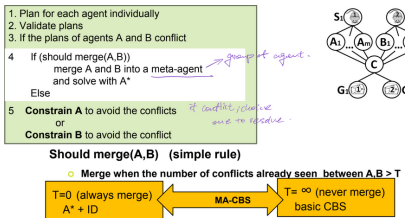
The weight for a pair of dependent agents is at least one

IDCBS: Replace the high-level search with IDA*

Independence Detection (ID)

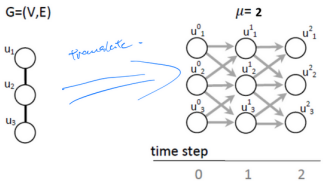
Simple Independence Detection
1. Solve optimally each agent separately
2. While some agents conflict
   1. Merge conflicting agents to one group
   2. Solve optimally new group

Meta Agent



MAPF via Reductions:
If anybody improves the solver for Q then we get an improved solver for P for free
Staying on the shoulders of giants
SAT model is expressed as a CNF formula We can go beyond CNF and use abstract expressions that are translated to CNF
Uses multi-valued state variables (logarithmic encoding) encoding position of agents in layers.

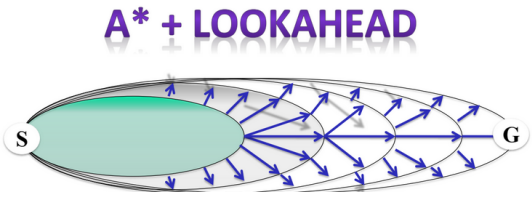
Time-Expanded Graph (TEG)



Use an encoding that finds a minimal-sum-of-costs solution for a given makespan
Find the optimal makespan μ∗ = μ0+Δ by trying Δ = 0, 1, 2, ... and the solution with minimal sum of costs within makespan μ∗(let this sum of costs be ξ∗)
Find the solution with minimal sum of costs within makespan μ0+ ξ∗- ξ0

Enhanced Techniques for A*
Lazy A*: When a node n is generated, LA*
- first computes h1(n), and adds n to OPEN.
- Only when n re-emerges as the top of OPEN, h2(n), evaluated.

Three types of nodes
(1) expanded regular (ER): nodes that were expanded after both heuristics were computed
(2) surplus regular (SR): nodes for which h2 was computed but are still in OPEN when the goal was found
(3) surplus good (SG): nodes for which only h1 was computed by LA* when the goal was found



Parameter to tune: cost bound or depth (for unit cost) of the lookahead
Memory: is saved with larger cost bounds
Time: larger cost bound give faster time per node but more duplicates

Enhanced Partial-Expansion A*:
When a solution is found with f=C nodes with f>C in OPEN are discarded
These are designated as surplus nodes
When expanding p with F(p)=K, EPEA* only generates children c with f(c)=K.
Surplus nodes
• Let X be the number of nodes expanded

Partial Expansion A* (PEA*)

• PEA* generates all the children but is **selective in adding them** into OPEN

```
When a node p is expanded with F=K
{
   •Generate all its children
      •Add children with f=K into OPEN
      •Children with f ≠ K are discarded
   •p is added back to OPEN with the best
   child exceeding K.
}
```

Pro:Surplus nodes are generated but are never added to OPEN
Memory is saved
Con:Re-expansion and re-genearation of nodes

PE-IDA*
Regular IDA* can be viewed as
using basic partial expansion IDA*
With a threshold of T
children with I7 are generated and expanded
Children with f > T are generated and discarded.
(E)PEA* Summary
When applicable, especially in exponential domains --- substantial time savings
Limitations:
Can blow up OPEN in domains with many cycles such as polynomial domains

Origin of Heuristics

Heuristic from Pattern Databases (PDBs)

A different method for abstracting and relaxing to problem to get a simplified problem

Distances from abstract spaces are lower bounds for the original problem

For enhanced algorithms: large open-lists or transposition tables. They store nodes explicitly.

A pattern database (PDB) is a lookup table that stores solutions to all configurations of the sub-problem (patterns)

If you have k of admissible heuristics (e.g. PDBs), their max is also admissible, h=max(h1,h2«hk)

Values of disjoint databases can be added and are still admissible

Partition the disks into disjoint sets **two parts**

Cost splitting – split the costs among the different (non- disjoint) sub-problems.

• **Location based costs** – we only charge the pattern that moved into a special location.

One such mapping maps each permutation to its index in a lexicographic ordering of all such permutations

Dual lookups are possible when there is a symmetry between locations and objects:

• Each object is in only one location and each location occupies only one object

A heuristic is inconsistent if for some two nodes n and m |h(n)-h(m)| > dist(n,m)

Randomizing a Heuristic:

Alternatively, we can randomize which heuristic out of K to consult.• Admissible • Inconsistent

• Benefits: Only one look up. BPMX can be activated

Inconsistent Heuristics and BPMX

• Works great and natural for IDA* and for exponential domains

• Has potential for A* and polynomial domains but you have to be Careful • Node reopening • BPMX is tricky

True Distance Heuristics: Useful in domains that fit in memory

Especially for polynomial domains

• Useful where we need (optimal) solutions very fast. (e.g. real time games, GPS)

Differential heuristics

- Choose **k** *pivot* states
- Store shortest paths to each of these states from all **N** states.
- Memory: **k*N**
- Time: **k* N**

$$h_x(a, b) = | d(a, x) - d(b,x) |$$

pivot state · distance from a to x ·

Canonical heuristics

Choose K (out of N) canonical states

• Primary data: Store all pairs shortest paths among all these K states

• Secondary data: From each state (out of N) store 1) which canonical state is the closest and 2) distance to the closest canonical state

Border Heuristics

Abstract the domain into K disjoint (canonical) regions

• For example, divide the world into countries

• A border state is a state at one region which has a neighbor at another region

• The border heuristic keeps the shortest distance between borders of regions

Enhanced CBS

bi-level search algorithm that solves MAPF bounded- suboptimally, i.e., its solution cost is guaranteed to be at most w times optimal.

The low level runs a focal search to plan paths for single agents.

• The high level runs a focal search on the constraint tree to resolve collisions.

Limitations of ECBS

First issue – the lower bound value lb($best_{lb}$) of ECBS rarely increases.

Second issue – ECBS ignores the potential cost increase below a node.

EECBS replaces the focal search with EES [Thayer and Ruml 2011] on the high level.

- ECBS has two limitations:
  - The lower bound value $lb(best_{lb})$ of ECBS rarely increases.
  - ECBS ignores the potential cost increase below a node.
- We proposed EECBS, a new bounded-suboptimal MAPF algorithm.
  1. We replaced focal search with EES on the high level;
  2. We learned informed heuristics for the high level; and
  3. We incorporated four CBS improvements.
- Empirically,
  - EECBS significantly outperforms the state-of-the-art bounded suboptimal MAPF algorithms ECBS, BCP, and eMDD-SAT; and
  - It can find solutions that are provably at most 2% worse than optimal for large MAPF instances with up to 1,000 agents.