Query `?- max(2,5,M).` yields `M=5` but on backtracking also tries second clause and fails or yields wrong.

With cut:

prolog Copy Edit

```prolog
max(X,Y,Y) :- X =< Y, !.    % if X =< Y, commit here
max(X,Y,X) :- X >  Y.        % only reached if first clause or its guard fails
```

Now:

1. If `2 =< 5` is true, cut prevents trying the second clause → one solution `M=5`.
2. If `X =< Y` fails (e.g. `X=7, Y=3`), first clause fails before cut, then second clause applies → `M=7`.

Q.1(b) Cuts in Prolog ("!"): concept & example [4 marks][CO5]

Answer:

A **cut** (`!`) is a procedural device that prunes Prolog's search tree: once execution passes a cut, **no alternative** clauses or choice-points created **before** that cut will be reconsidered on backtracking.

- Cuts are used to commit to decisions and improve efficiency or to encode "if-then-else" behavior.
- There are two kinds:
  - **Green cut**: does not affect logical correctness, only performance.
  - **Red cut**: affects which solutions are found (can change program meaning).

Example: max of two numbers

Without cut, you get two solutions:

prolog Copy Edit

```prolog
max(X,Y,Y) :- X =< Y.
max(X,Y,X) :- X >  Y.
```

Query `?- max(2,5,M).` yields `M=5` but on backtracking also tries second clause and fails or yields wrong.

With cut:

Example program: find a member of a list

prolog Copy Edit

```prolog
% member(X, List) is true if X is one of the elements of List.
member(X, [X|_]).          % clause 1: X is head
member(X, [_|Tail]) :-
    member(X, Tail).       % clause 2: or X is in the tail
```

Demonstration of backtracking:

text Copy Edit

```text
?- member(X, [a,b,c]).
```

1. Try clause 1: unify X=a → success, yields `X = a`.
2. On user request for more solutions, backtrack to the choice point in clause 2.
3. Clause 2 calls `member(X, [b,c])`: again clause 1 gives `X = b`.
4. Backtracking again yields `X = c`.
5. Finally no more elements → overall failure and stop.

Thus Prolog systematically explores all ways (`a`, then `b`, then `c`) by backtracking.

Q.1(a) What is backtracking in Prolog? Explain with a program. [4 marks][CO5]

Answer:

Backtracking is Prolog's built-in mechanism for exploring alternative ways to satisfy a goal.

1. When Prolog attempts a goal (predicate call), it tries clauses top-to-bottom and the body of each clause left-to-right.
2. If a subgoal fails, Prolog **backs up** ("backtracks") to the most recent choice point (a place where another clause or another choice remained) and tries the next alternative.
3. This continues until either the goal succeeds (with some variable bindings) or all alternatives are exhausted (and the query fails).

Example program: find a member of a list

prolog Copy Edit

```prolog
% member(X, List) is true if X is one of the elements of List.
member(X, [X|_]).          % clause 1: X is head
member(X, [_|Tail]) :-
    member(X, Tail).       % clause 2: or X is in the tail
```

Simple example (4-city TSP)

Cities: A, B, C, D; distances given.

- **Iteration 1**: ants roam nearly at random → generate tours like A-C-B-D (length 10), B-D-A-C (length 12), etc.
- **Pheromone update**: shorter tours deposit more pheromone (e.g. edges in A-C-B-D get larger $\Delta\tau$).
- **Iteration 2**: ants more likely follow heavily "marked" edges (e.g. A→C, C→B, B→D).
- After several iterations, most ants converge on the optimal or near-optimal tour, say A-C-D-B.

Why it works

- **Positive feedback** (good solutions reinforced)
- **Distributed** (multiple ants explore in parallel)
- **Exploration vs. exploitation** balanced by evaporation

Algorithm sketch for TSP

1. **Initialization**
   - Place m ants on random cities.
   - Initialize pheromone $\tau_{ij} = \tau_0$ on every edge (i, j).
2. **Solution construction** (each ant builds a tour)
   - From city i, ant chooses next city j not yet visited with probability

     $$P_{ij} = \frac{\tau_{ij}^{\alpha}\,\eta_{ij}^{\beta}}{\sum_{k\in\text{allowed}} \tau_{ik}^{\alpha}\,\eta_{ik}^{\beta}},$$

     where $\eta_{ij} = 1/d_{ij}$ is the inverse distance, and α, β control pheromone vs. heuristic weight.
3. **Pheromone update**
   - **Evaporate**: $\tau_{ij} \leftarrow (1-\rho)\,\tau_{ij}$ for all edges, with $0 < \rho < 1$.
   - **Deposit**: each ant k deposits $\Delta\tau_{ij}^{k} = Q/L_k$ on edges it used, where L_k is its tour length.
   - New pheromone: $\tau_{ij} \leftarrow \tau_{ij} + \sum_k \Delta\tau_{ij}^{k}$.
4. **Repeat** until stopping criterion (max iterations or convergence).

Ant Colony Optimization (ACO) is a population-based metaheuristic inspired by how real ants find shortest paths via pheromone trails.

Core ideas

1. **Artificial ants** construct solutions step by step, making probabilistic choices biased by "pheromone" levels on solution components.
2. After all ants build solutions, **pheromone update** reinforces components used in the best solutions and evaporates all pheromones slightly to avoid premature convergence.
3. Over iterations, good components accumulate more pheromone → ants converge on high-quality solutions.

Algorithm sketch for TSP

1. **Initialization**
   - Place m ants on random cities.
   - Initialize pheromone $\tau_{ij} = \tau_0$ on every edge (i, j).

Example: Final Plan

- The final plan has the following components:
- Actions: {RightSock, RightShoe, LeftSock, LeftShoe, Start, Finish}
- Orderings: {RightSock ‹ RightShoe, LeftSock ‹ LeftShoe}
- Open preconditions: {}
- Links:

$$RightSock \xrightarrow{RightSockOn} RightShoe$$

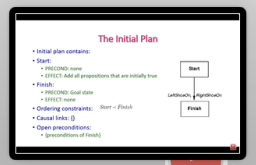
$$LeftSock \xrightarrow{LeftSockOn} LeftShoe$$

$$RightShoe \xrightarrow{RightShoeOn} Finish$$

$$LeftShoe \xrightarrow{LeftShoeOn} Finish$$

Next…

- Successor function
  - Arbitrarily picks one open precondition p on an action B and generates a successor plan, for every possible consistent way of choosing an action A, that achieves p
- Consistency:
  - Causal link $A \xrightarrow{p} B$ and the ordering constraint are added ( ) $(A \prec B \quad Start \prec A \quad A \prec Finish)$
  - Resolve conflict: add $B \prec C$ or $C \prec A$

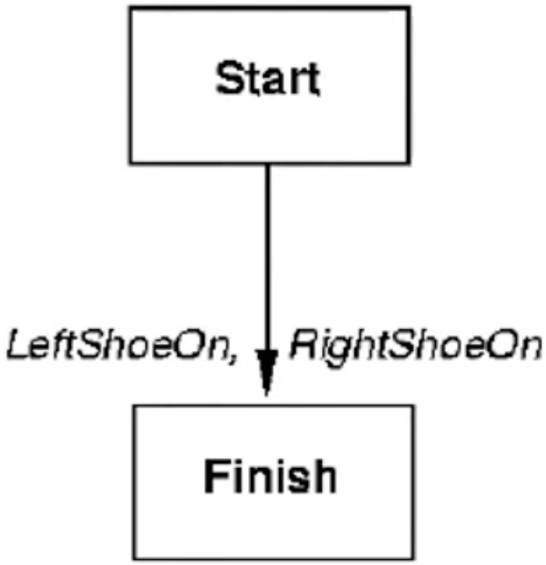


Subscribe

5:15 / 7:42

The Initial Plan

- Initial plan contains:
- Start:
  - PRECOND: none
  - EFFECT: Add all propositions that are initially true
- Finish:
  - PRECOND: Goal state
  - EFFECT: none
- Ordering constraints: $Start \prec Finish$
- Causal links: {}
- Open preconditions:
  - {preconditions of Finish}

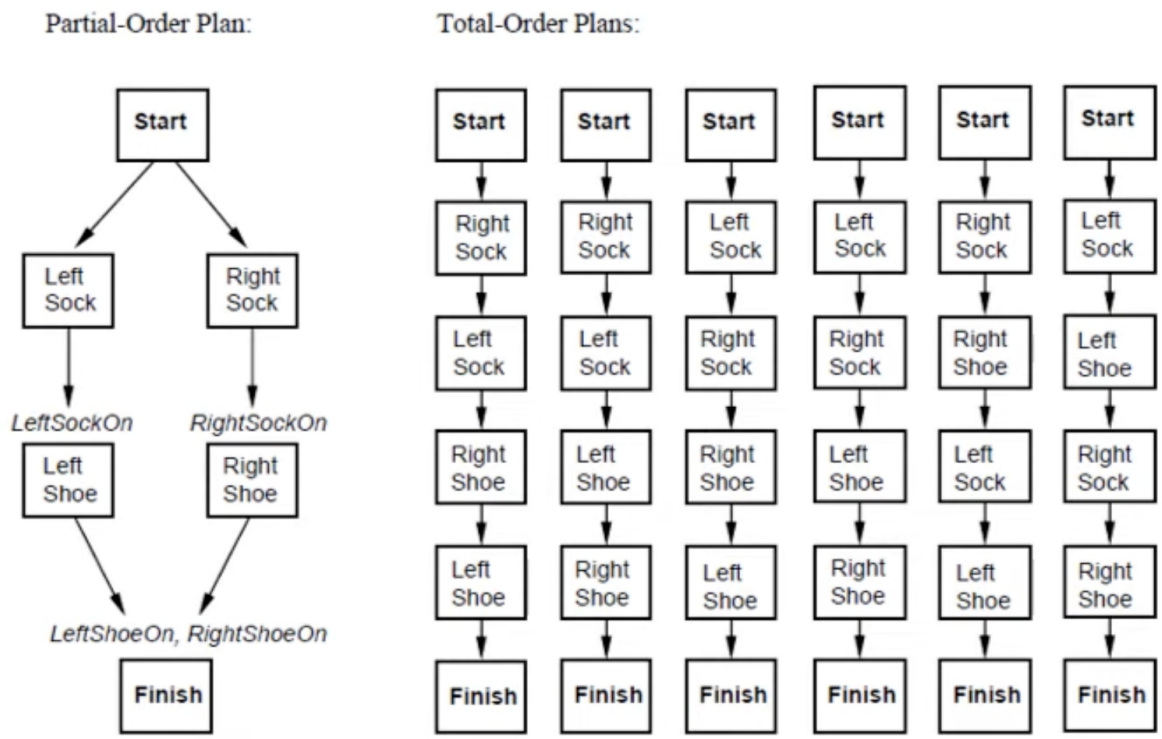


How to Define Partial Order Plan?

- A set of actions, that make up the steps of the plan
- A set of ordering constrain $A \prec B$
  - A before B
- A set of causal links: $A \xrightarrow{p} B$
  - A achieves P for B $RightSock \xrightarrow{RightSockOn} RightShoe$
  - May be conflicts if C has the effect of ¬P and if C comes after A and before B
- A set of open preconditions:
  - A precondition is open, if it is not achieved by some action in the plan

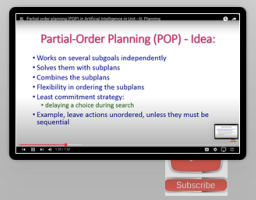
The partial-order plan - The shoes and socks problem

- A partial-order plan for putting on shoes and socks, and the six corresponding linearizations into total-order plans



POP Example - Putting on a pair of shoes:

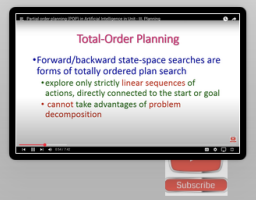
- Goal(RightShoeOn ^ LeftShoeOn)
- Init()
- Action: RightShoe
  - PRECOND: RightSockOn
  - EFFECT: RightShoeOn
- Action: RightSock
  - PRECOND: None
  - EFFECT: RightSockOn
- Action:LeftShoe
  - PRECOND: LeftSockOn
  - EFFECT: LeftShoeOn
- Action: LeftSock
  - PRECOND: None
  - EFFECT: LeftSockOn



2:02 / 7:42

Partial-Order Planning (POP) - Idea:

- Works on several subgoals independently
- Solves them with subplans
- Combines the subplans
- Flexibility in ordering the subplans
- Least commitment strategy:
  - delaying a choice during search
- Example, leave actions unordered, unless they must be sequential



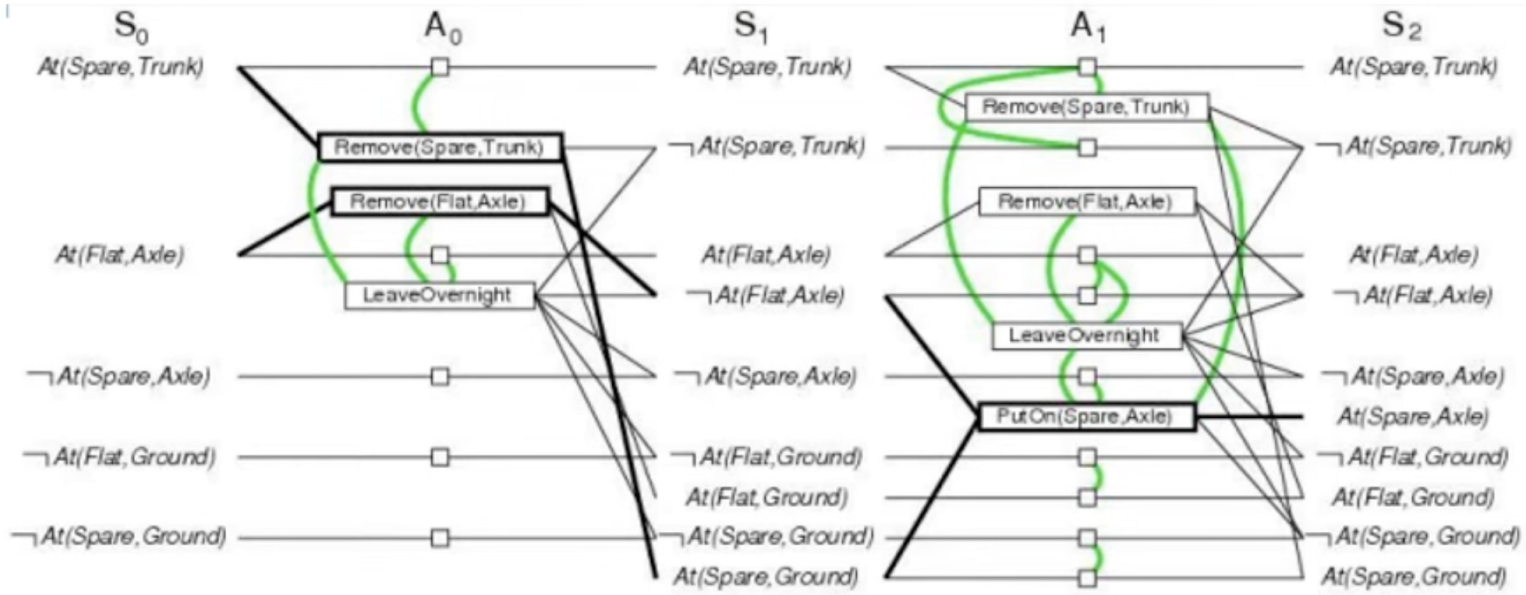
1:39 / 7:42

Total-Order Planning

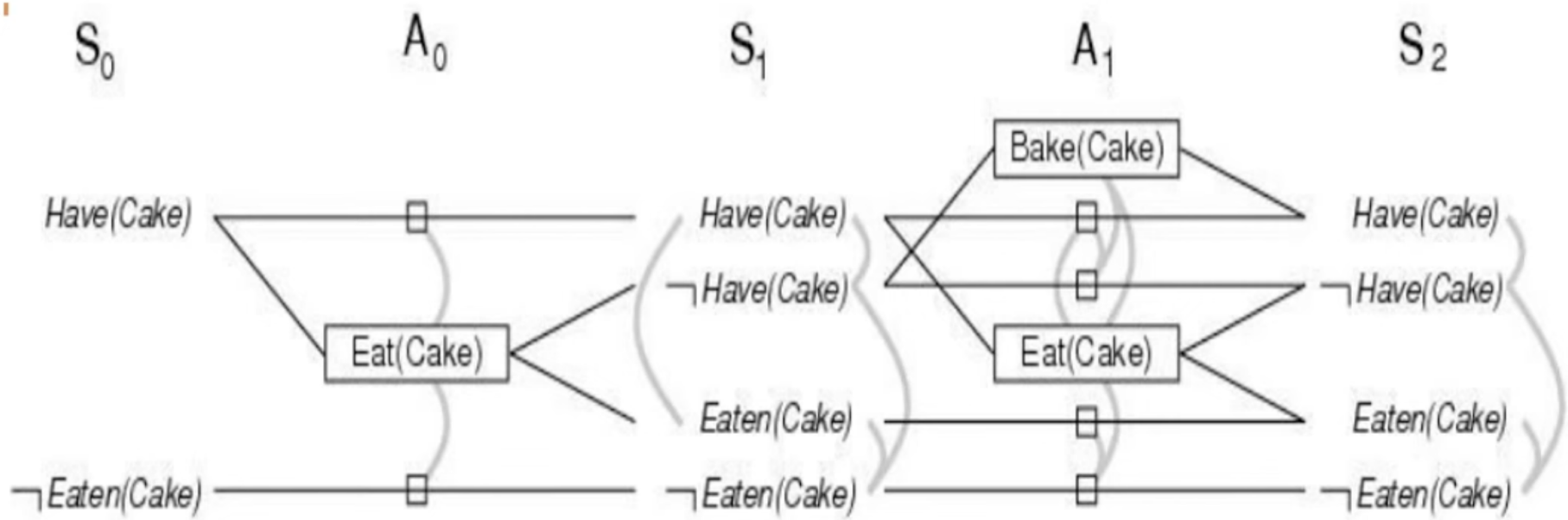
- Forward/backward state-space searches are forms of totally ordered plan search
  - explore only strictly linear sequences of actions, directly connected to the start or goal
  - cannot take advantages of problem decomposition

0:54 / 7:42

GRAPHPLAN example – Change Flat Tire



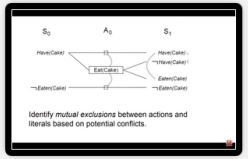
- Initially the plan consist of 5 literals from the initial state (S0).
- Add actions whose preconditions are satisfied by EXPAND-GRAPH (A0)
- Also add persistence actions and mutex relations.
- Add the effects at level S1
- Repeat until goal is in level Si

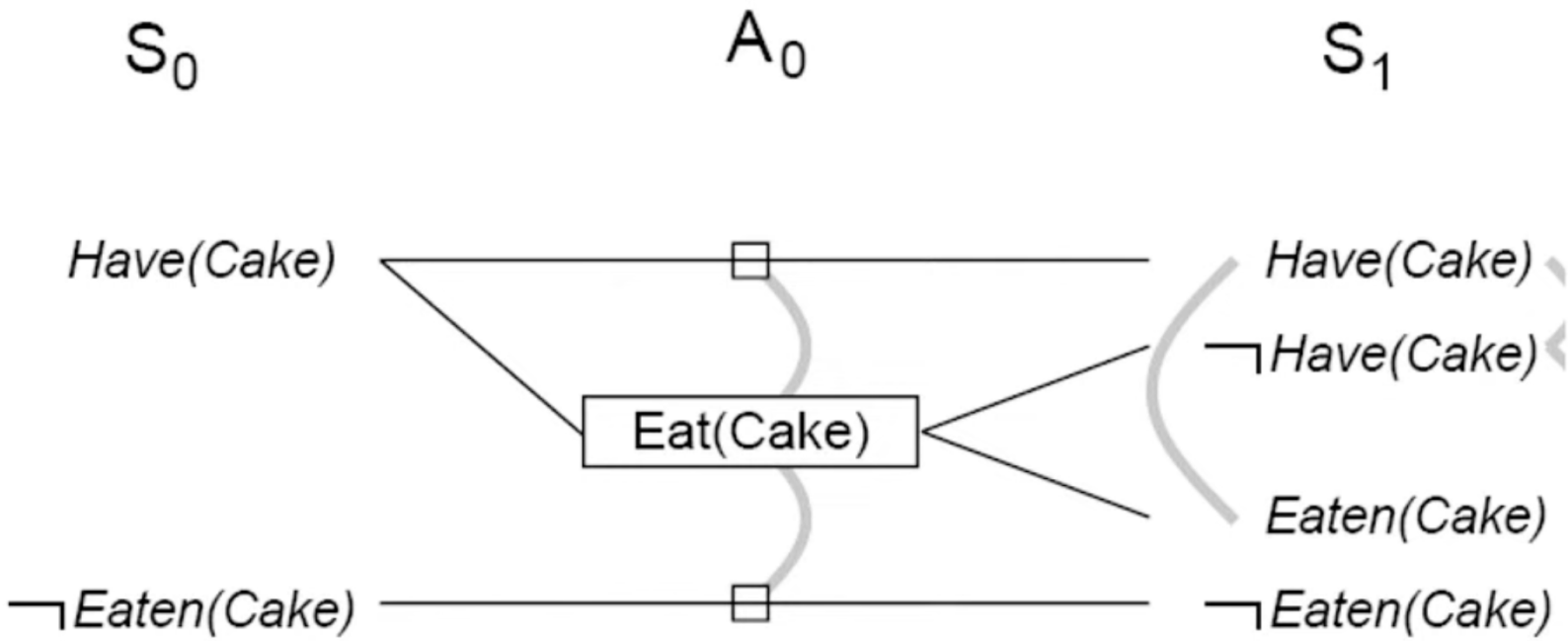


- Repeat process until graph levels off:
  - two consecutive levels are identical, or
  - contain the same amount of literals

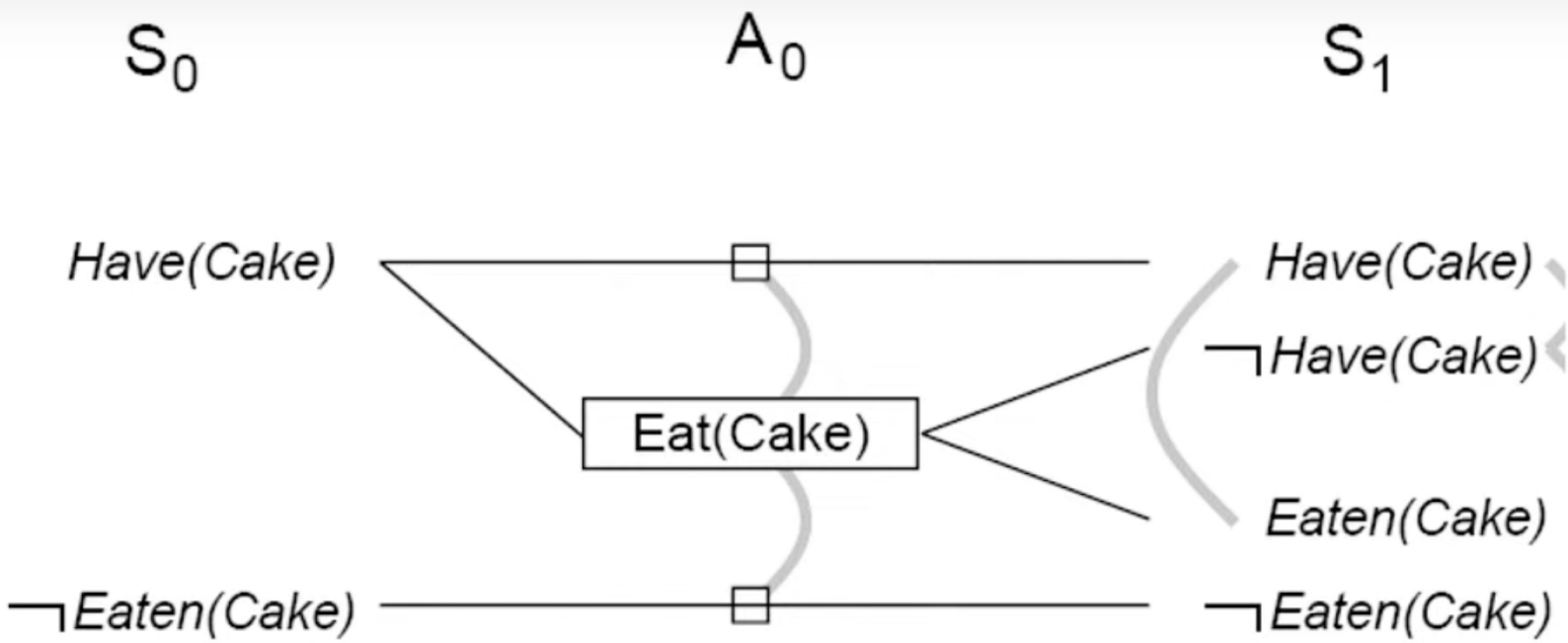
Mutual exclusion

- A mutex relation holds between two actions when:
  - Inconsistent effects: one action negates the effect of another.
  - Interference: one of the effects of one action, is the negation of a precondition of the other.
  - Competing needs: one of the preconditions of one action, is mutually exclusive with the precondition of the other.
- A mutex relation holds between two literals when:
  - one is the negation of the other
  - each possible action pair that could achieve the literals is mutex (inconsistent support).





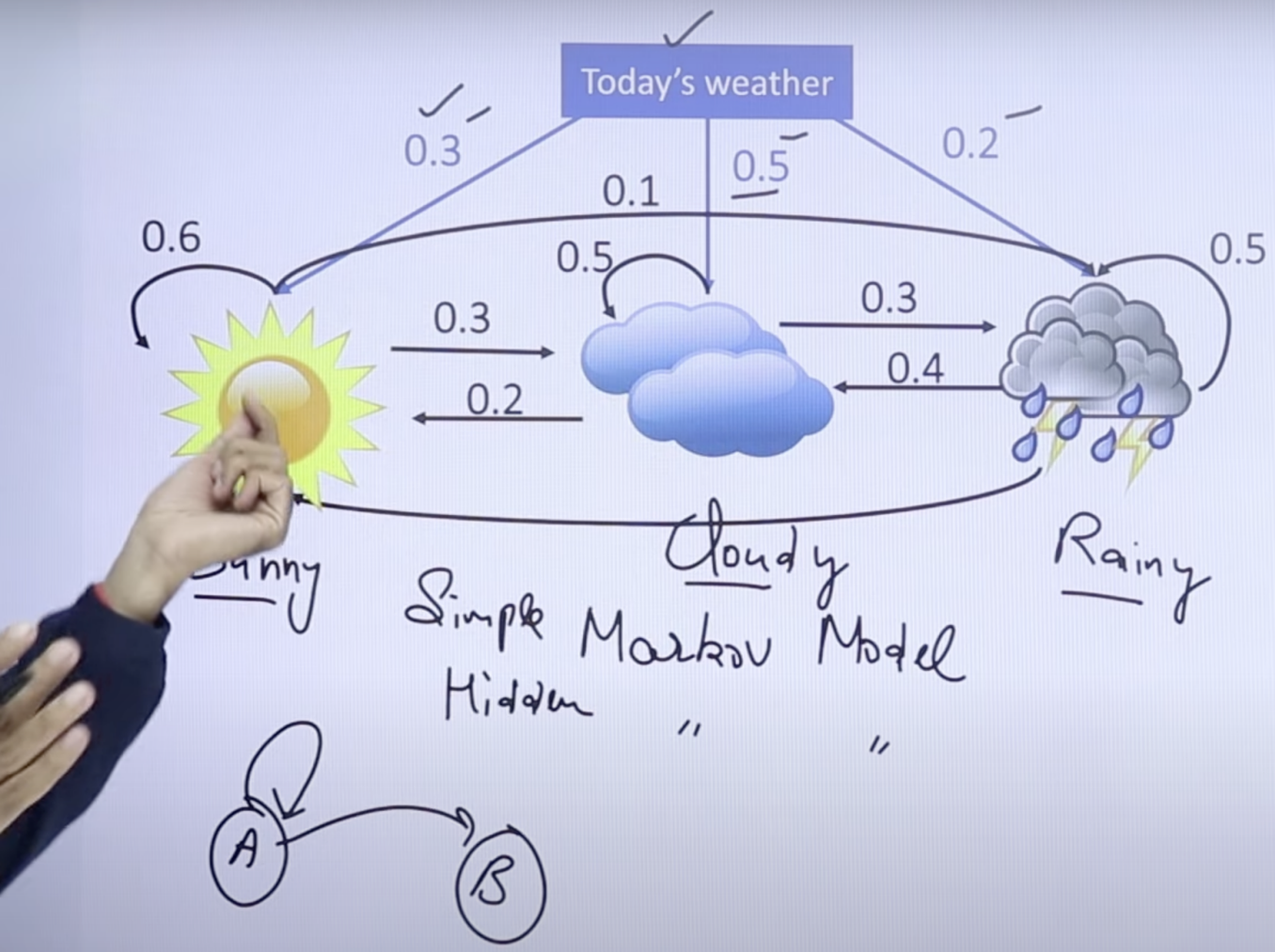
Identify *mutual exclusions* between actions and literals based on potential conflicts.



Identify *mutual exclusions* between actions and literals based on potential conflicts.

6:30 / 13:10

Uncertainty probabilistic inference (Markov Model) | Artificial Intelligence



Sunny	Cloudy	Rainy
0.3	0.5	0.2

	Sunny	Cloudy	Rainy
Sunny	0.6	0.3	0.1
Cloudy	0.2	0.5	0.3
Rainy	0.1	0.4	0.5

SUBSCRIBE

5:00 / 14:14

3. Summary of Differences

Feature	Simple MC	Hidden Markov Model
State visibility	Directly observable	Hidden, must be inferred
Model parameters	π, A	π, A, B
Primary algorithms	multiply A or diagonalize	Forward/Backward, Viterbi, Baum–Welch
Use cases	Queues, reliability, games	Speech recognition, bioinformatics, NLP

Key takeaway:

- A **simple Markov chain** models a sequence of fully observed states with the Markov property.
- A **Hidden Markov Model** wraps a latent Markov chain inside an emission model; you only see noisy outputs and must solve inference (decoding) and learning problems on top of state transitions.

2.4 Example: Part-of-Speech Tagging

- Hidden states = POS tags (NOUN, VERB, etc.).
- Observations = words in a sentence.
- Transitions model how likely one tag follows another.
- Emissions model how likely a tag generates a given word.

Given a sentence ("Time flies like an arrow"), Viterbi finds the most probable tag sequence.

2.5 Comparison with simple Markov chain

Aspect	Simple MC	HMM
States observable?	Yes	No (hidden)
Observations	N/A	Emissions dependent on hidden state
What you infer	Next-state probabilities	Both hidden states and observations
Algorithms	Matrix powers, eigenvalues	Forward/Backward, Viterbi, EM

Graphical plate:

```scss
X0 → X1 → X2 → …      (hidden)
 |     |     |
 v     v     v
O0     O1    O2 …      (observed)
```

2.3 Key computational problems

1. **Evaluation**: given model (A, B, π) and observation sequence $O_{0:T}$, compute $P(O_{0:T})$.
   - Solved by the **forward algorithm** in $O(N^2T)$.
2. **Decoding**: find the most likely hidden-state sequence $X_{0:T}$ given observations.
   - Solved by the **Viterbi algorithm** in $O(N^2T)$.
3. **Learning**: given only observations, estimate A, B, π.
   - Solved by the **Baum–Welch** (EM) algorithm.

2.4 Example: Part-of-Speech Tagging

2. Hidden Markov Model (HMM)

2.1 Motivation

In many real problems the true "state" of the system is not directly visible. Instead you see some **emission** (observation) that depends probabilistically on the hidden state.

2.2 Definition

An HMM augments a Markov chain with an observation process. It comprises:

1. **Hidden states** $S = \{s_1, \ldots, s_N\}$.
2. **Observations** $O = \{o_1, \ldots, o_M\}$.
3. **Initial state distribution** $\pi_i = P(X_0 = s_i)$.
4. **State-transition matrix** $A = [a_{ij}]$, as before.
5. **Emission matrix** $B = [b_i(o)]$, where

$$b_i(o_k) = P(\, O_t = o_k \mid X_t = s_i \,), \quad \sum_{k=1}^{M} b_i(o_k) = 1 \; \forall i.$$

1.3 Evolution

If at time t the chain is in state s_i, then at time $t+1$ it moves to state s_j with probability a_{ij}. After t ste
the distribution over states is

$$\pi^{(t)} = \pi^{(0)} A^t.$$

1.4 Example: Weather

State	Interpretation
S	Sunny
R	Rainy

Transition matrix:

$$A = \begin{pmatrix} P(S \to S) & P(S \to R) \\ P(R \to S) & P(R \to R) \end{pmatrix} = \begin{pmatrix} 0.8 & 0.2 \\ 0.5 & 0.5 \end{pmatrix}.$$

If today is Sunny, tomorrow is Sunny with prob. 0.8, Rainy with 0.2, etc.

1.5 When to use

– You can **observe** the state directly at each time step.
– You care only about how states evolve over time (no hidden structure).
– Applications: board-game positions, simple queui systems, population models.

1. Simple Markov Model (Markov Chain)

1.1 Definition

A (time-homogeneous) Markov chain is a sequence of random variables

$$X_0, X_1, X_2, \ldots$$

taking values in a finite (or countable) state set $S = \{s_1, \ldots, s_N\}$, such that for every t and every s
$i, j \in S$:

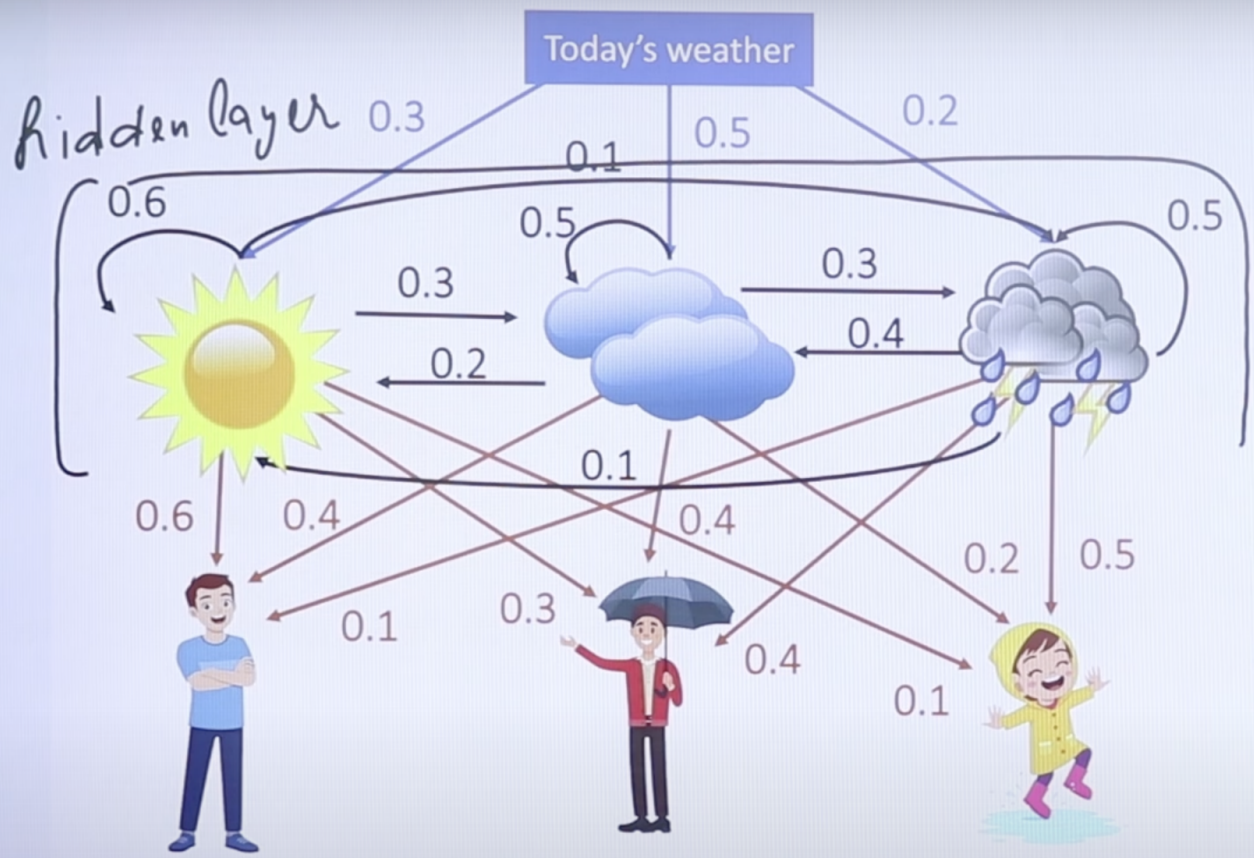
$$P(X_{t+1} = s_j \mid X_t = s_i,\ X_{t-1}, \ldots, X_0) \ = \ P(X_{t+1} = s_j \mid X_t = s_i)$$

—that is, the next state depends only on the current state (the **Markov property**).

1.2 Components

1. **State space** $S = \{s_1, \ldots, s_N\}$.
2. **Initial distribution** $\pi_i = P(X_0 = s_i)$.
3. **Transition matrix**

$$A = [a_{ij}], \quad a_{ij} = P(X_{t+1} = s_j \mid X_t = s_i), \quad \sum_{j=1}^{N} a_{ij} = 1 \ \forall i.$$

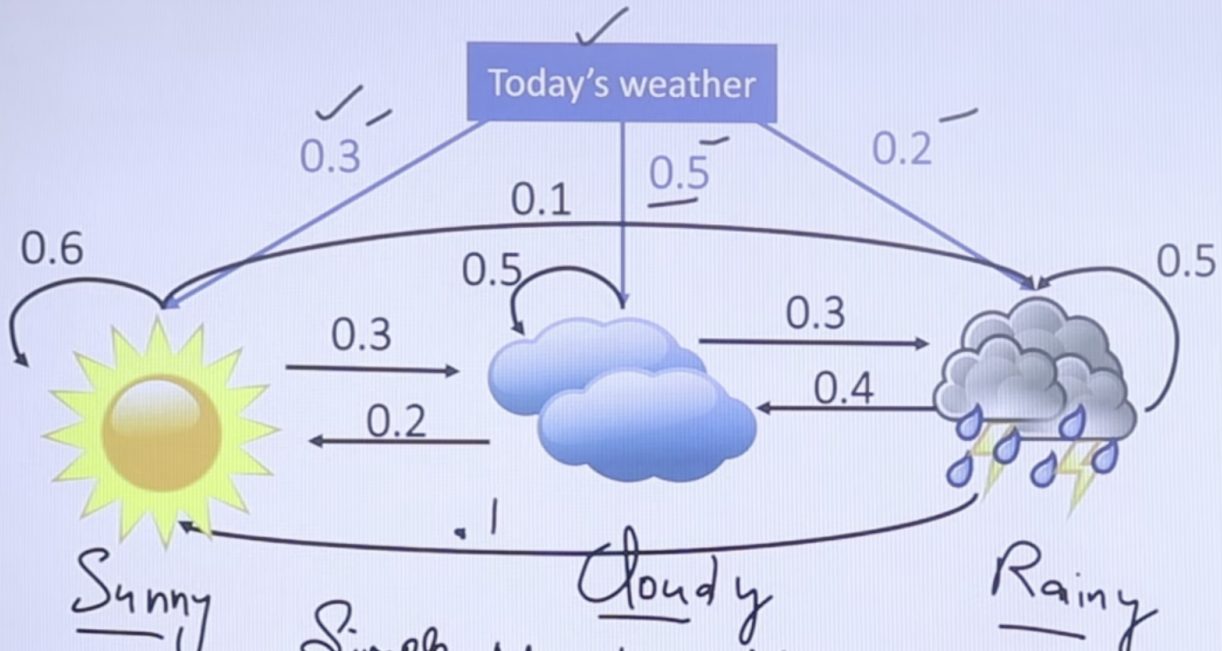


Sunny	Cloudy	Rainy
0.3	0.5	0.2

	Sunny	Cloudy	Rainy
Sunny	0.6	0.3	0.1
Cloudy	0.2	0.5	0.3
Rainy	0.1	0.4	0.5

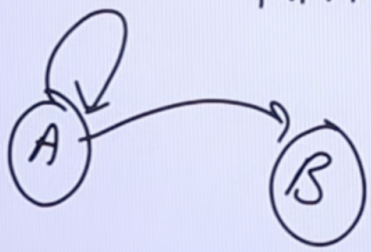
	Normal	Umbrella	Raincoat
Sunny	0.6	0.3	0.1
Cloudy	0.4	0.4	0.2
Rainy	0.1	0.4	0.5

1:05 / 13:48



Simple Markov Model

Hidden " "



Sunny	Cloudy	Rainy
0.3	0.5	0.2

tom.

	Sunny	Cloudy	Rainy
Sunny	0.6	0.3	0.1
Cloudy	0.2	0.5	0.3
Rainy	0.1	[illegible]4	0.5

PS

ChatGPT
Sora
Library
Explore GPTs

Previous 7 Days
Law of Variable Proportions
Feasible Region Markowitz Mo
Feynman-Kac Theorem Explar

Previous 30 Days
Plan Space Planning Overview

April
Chess Evaluation Functions

February
Web Dev Internship Update
Internship Experience Summar
ISTernship Fellowship Statem
ISTernship Statement Purpose

Upgrade plan
More access to the best models

simplifying duplicate checks and supporting directed graphs without forbidden-operator overhead.

IJCAI

6. When to use SMGS

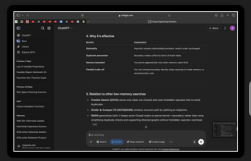
- Very large state-spaces where Closed would not fit in memory.
- Problems with many duplicate paths (where IDA*/RBFS perform badly).
- You still need guaranteed optimal solutions.

In a nutshell, Sparse-Memory Graph Search prunes away most of the Closed list—leaving just a "skin" that (a) prevents re-expansion and (b) preserves enough back-pointers to reconstruct the optimal path via a small divide-and-conquer step—thus achieving optimal graph search in bounded memory.

Sources

Follow up

↳ How does SMGS handle memory pruning trade-offs?



Ask anything

Search | Reason | Deep research | Create image

ChatGPT can make mistakes. Check important info. See Cookie Preferences.

ChatGPT
Sora
Library
Explore GPTs

Previous 7 Days
Law of Variable Proportions
Feasible Region Markowitz Mo
Feynman-Kac Theorem Explar

Previous 30 Days
Plan Space Planning Overview

April
Chess Evaluation Functions

February
Web Dev Internship Update
Internship Experience Summar
ISTernship Fellowship Statem
ISTernship Statement Purpose

Upgrade plan
More access to the best models

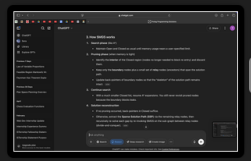
4. Why it's effective

Benefit	Explanation
Optimality	Heuristic remains admissible/consistent; search order unchanged.
Duplicate prevention	Boundary nodes suffice to block all back-leaks.
Memory bounded	You prune aggressively only when memory nears limit.
Flexible trade-off	You can choose boundary density (relay spacing) to trade memory vs. reconstruction cost.

5. Relation to other low-memory searches

- **Frontier Search (DCFS)** stores only Open (no Closed) and uses forbidden-operator lists to avoid duplicates.
- **Divide-&-Conquer FS (DCFS/DCBS)** similarly recovers path by splitting at midpoints.
- **SMGS** generalizes both: it keeps *some* Closed nodes (a sparse kernel + boundary) rather than none, simplifying duplicate checks and supporting directed graphs without forbidden-operator overhead.

IJCAI



Ask anything
Search
Reason
Deep research
Create image
ChatGPT can make mistakes. Check important info. See Cookie Preferences.

ChatGPT
Sora
Library
Explore GPTs

Previous 7 Days
Law of Variable Proportions
Feasible Region Markowitz Mo
Feynman-Kac Theorem Explar

Previous 30 Days
Plan Space Planning Overview

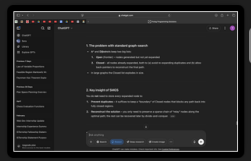
April
Chess Evaluation Functions

February
Web Dev Internship Update
Internship Experience Summar
ISTernship Fellowship Statem
ISTernship Statement Purpose

Upgrade plan
More access to the best models

3. How SMGS works

1. **Search phase** (like A*)
   - Maintain Open and Closed as usual until memory usage nears a user-specified limit.
2. **Pruning phase** (when memory is tight)
   - Identify the **interior** of the Closed region (nodes no longer needed to block re-entry) and discard them.
   - Keep only the **boundary** nodes plus a small set of **relay** nodes (ancestors) that span the solution path.
   - Update back-pointers of boundary nodes so that the "skeleton" of the solution path remains intact. IJCAI
3. **Continue search**
   - With a much smaller Closed list, resume A* expansions. You still never revisit pruned nodes because the boundary blocks leaks.
4. **Solution reconstruction**
   - If no pruning occurred, back-pointers in Closed suffice.
   - Otherwise, extract the **Sparse Solution Path (SSP)** via the remaining relay nodes, then recursively re-solve each gap by re-invoking SMGS on the sub-graph between relay nodes (divide-and-conquer). IJCAI



Ask anything
Search
Reason
Deep research
Create image
ChatGPT can make mistakes. Check important info. See Cookie Preferences.

ChatGPT
Sora
Library
Explore GPTs

Previous 7 Days
Law of Variable Proportions
Feasible Region Markowitz Mo
Feynman-Kac Theorem Explar

Previous 30 Days
Plan Space Planning Overview

April
Chess Evaluation Functions

February
Web Dev Internship Update
Internship Experience Summar
ISTernship Fellowship Stateme
ISTernship Statement Purpose

Upgrade plan
More access to the best models

1. The problem with standard graph-search

- **A*** and **Dijkstra's** keep two big lists:
  1. **Open** (frontier) – nodes generated but not yet expanded
  2. **Closed** – all nodes already expanded, both to (a) avoid re-expanding duplicates and (b) allow back-pointers to reconstruct the final path.
- In large graphs the Closed list explodes in size.

2. Key insight of SMGS

You do **not** need to store *every* expanded node to:

1. **Prevent duplicates** – it suffices to keep a "boundary" of Closed nodes that blocks any path back into fully closed regions.
2. **Reconstruct the solution** – you only need to preserve a sparse chain of "relay" nodes along the optimal path; the rest can be recovered later by divide-and-conquer. IJCAI

Ask anything
Search
Reason
Deep research
Create image
ChatGPT can make mistakes. Check important info. See Cookie Preferences.

3. Breadth-First Heuristic Search (BFHS)

3.1 Intuition

Standard **BFS** expands strictly by increasing depth, ignoring any heuristic. **Heuristic search** (A*, greedy) orders purely by estimated cost.

Breadth-First Heuristic Search blends them:

- You still expand in order of **increasing depth** (like BFS) to guarantee you find the shallowest solution.
- **Within** each depth-level, you use a heuristic to choose which nodes to expand first and/or which to prune if there are too many.

This gives you:

- **Completeness** and **optimality** (when shallowest = best) of BFS
- Some of the **focus** of heuristic guidance

2. Divide-and-Conquer Beam Search

2.1 Intuition

When the problem naturally **decomposes** into sub-parts (e.g. sequence labeling, machine translation), you:

1. **Split** the problem into two (or more) smaller sub-problems
2. **Beam-search** each sub-problem independently
3. **Merge** the best partial solutions to form candidates for the full problem

By limiting beam width at each sub-stage, you keep each search tractable; by dividing you avoid an explosion in partial hypotheses.

1. Beam-Stack Search

1.1 Intuition

- **Beam-search** keeps only the top k partial solutions (the “beam”) at each depth, discarding the rest.
- **Depth-first with stack** allows backtracking when a beam “dies” (no further expansions).

Beam-Stack search marries the two: you do a depth-first exploration, but at each depth you only push onto the stack your top k successors. If you reach a dead end, you pop back up (backtrack) and resume with the next best beam choice.

(b) (ii) Split a list into two approximately equal halves

prolog

Copy Edit

```prolog
% split_list(List, L1, L2)
split_list(L, L1, L2) :-
  length(L, Len),
  Half is Len // 2,
  split_at(Half, L, L1, L2).

% split_at(N, List, Front, Back)
split_at(0, L, [], L).
split_at(N, [H|T], [H|F], B) :-
  N > 0,
  N1 is N - 1,
  split_at(N1, T, F, B).
```

(a) Sum of first N natural numbers

prolog Copy Edit

```prolog
% sum_n(N, S) succeeds if S is the sum 1+2+…+N.
sum_n(0, 0).
sum_n(N, S) :-
    N > 0,
    N1 is N - 1,
    sum_n(N1, S1),
    S  is S1 + N.
```

(b) (i) Sum of all elements of a given list

prolog Copy Edit

```prolog
% sum_list(List, Sum)
sum_list([], 0).
sum_list([H|T], Sum) :-
  sum_list(T, Rest),
  Sum is H + Rest.
```