University of Strathclyde Science

Lecture 4: Uninformed Search

Dr John Levine

CS310 Foundations of Artificial Intelligence
February 2nd 2016

Problem Solving using Search

Many problems in AI involve *deliberative reasoning*, leading to search in very big implicitly-defined graphs:

- Route finding in robotics
- Blocks World planning
- Rubik's cube
- Logistics planning
- Task scheduling
- Data mining
- Machine learning

What are Problems?

Each of these problems can be characterised by:

- Problem states, including the start state and the goal state
- Legal moves, or actions which transform problem states into other states
- Example: Rubik’s cube
- The start state is the muddled up cube, the goal is to have the state in which all sides are the same colour and the moves are the rotations of sides of the cube

Solutions

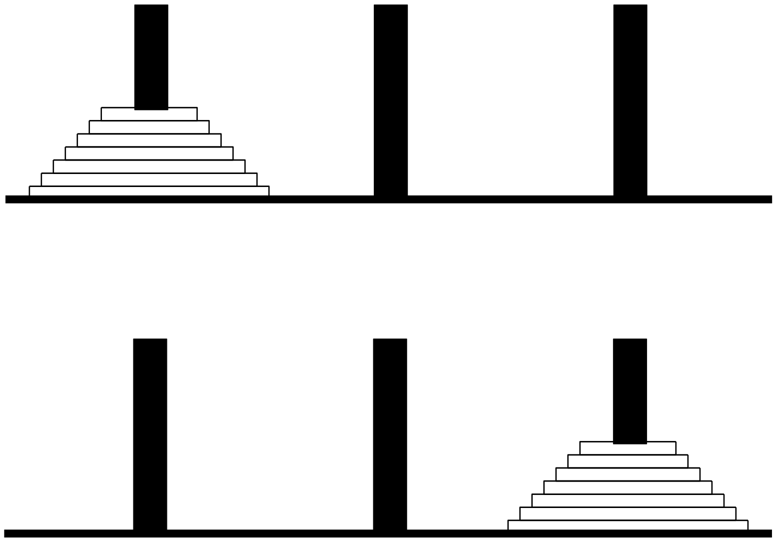
- Solutions are sequences of moves which transform the start state into the goal state
- The quality of the solution required will affect the amount of work we need to do
  - any solution will do
  - fixed amount of time, return best solution
  - near optimal solution needed
  - optimal solution needed

Formulating Problems

- A good formulation saves work
  - less search for the answer
- Three requirements for a search algorithm:
  - formal structures to describe the states
  - rules for manipulating them
  - identifying what constitutes a solution
- This gives us a state space representation

State Space Representation

- A state space comprises
  - states: snapshots of the problem
  - operators: how to move from one state to another



Example problem: Towers of Hanoi

Only move one disc at a time

Never put a larger disc on top of a smaller one

State Space Search

Problem solving using state space search consists of the following four steps:

1. Design a representation for states (including the initial state and the goal state)
2. Characterise the operators
3. Build a goal state recogniser
4. Search through the state space somehow by considering (in some or other order) the states reachable from the initial and goal states

Example: Blocks World

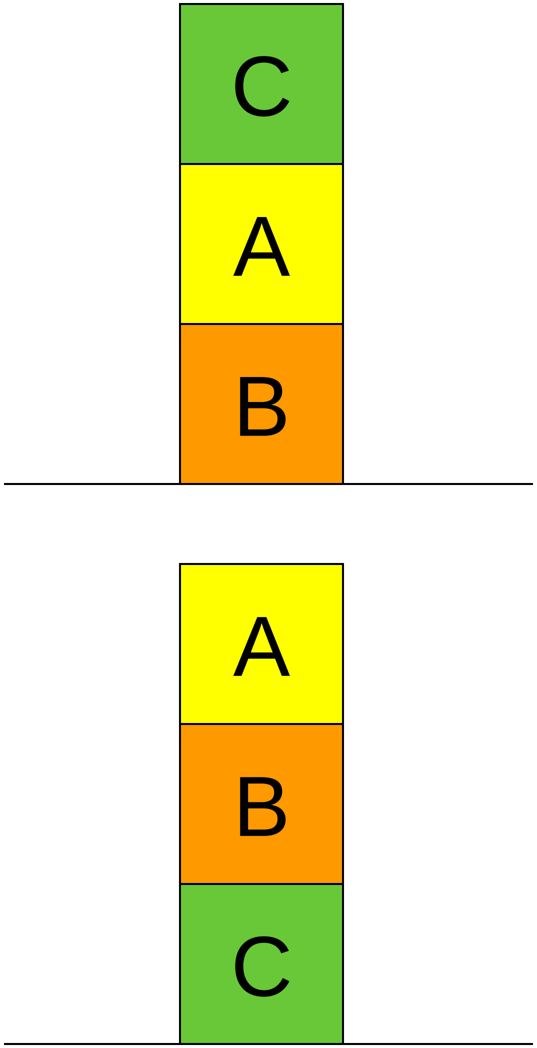
A "classic" problem in AI planning

The aim is to rearrange the blocks using the single robot arm so that the configuration in the goal state is achieved

An optimal solution performs the transformation using as few steps as possible

Any solution: linear complexity

Optimal solution: exponential complexity (NP hard)



Blocks World Representation

The blocks world problem can be represented as:

- States: stacks are lists, states are sets of stacks e.g. initial state = { [a,b],[c] }
- Transitions between states can be done using a single move operator: move(x,y) picks up object x and puts it on y (which may be the table)

  { [a,b,c] } → { [b,c],[a] }
  by applying move(a,table)

  { [a],[b,c] } → { [a,b,c] }
  by applying move(a,b)

Blocks World Representation

- NextStates(State) → list of legal states resulting from a single transition
  e.g. NextStates({ [a,b],[c] }) →
  { [a],[b],[c] } by applying move(a,table)
  { [b],[a,c] } by applying move(a,c)
  { [c,a,b] } by applying move(c,a)
- Goal(State) returns true if State is identical with the goal state
- Search the space: start with the start state, explore reachable states, continue until the goal state is found

Blocks World: NextStates Function

State	NextStates(State)
{ [a],[b],[c] }	{ [a,b],[c] }, { [a,c],[b] }, { [b,a],[c] }, { [b,c],[a] }, { [c,a],[b] }, { [c,b],[a] }
{ [a,b],[c] }	{ [a],[b],[c] }, { [a,c],[b] }, { [c,a,b] }
{ [a,c],[b] }	{ [a],[b],[c] }, { [a,b],[c] }, { [b,a,c] }
{ [b,a],[c] }	{ [a],[b],[c] }, { [b,c],[a] }, { [c,b,a] }
{ [b,c],[a] }	{ [a],[b],[c] }, { [b,a],[c] }, { [a,b,c] }
{ [c,a],[b] }	{ [a],[b],[c] }, { [c,b],[a] }, { [b,c,a] }
{ [c,b],[a] }	{ [a],[b],[c] }, { [c,a],[b] }, { [a,c,b] }
{ [a,b,c] }	{ [b,c],[a] }
{ [a,c,b] }	{ [c,b],[a] }
{ [b,a,c] }	{ [a,c],[b] }
{ [b,c,a] }	{ [c,a],[b] }
{ [c,a,b] }	{ [a,b],[c] }
{ [c,b,a] }	{ [b,a],[c] }

Formulating a Search Problem

Example: a truck moves around delivering packages

You will need:

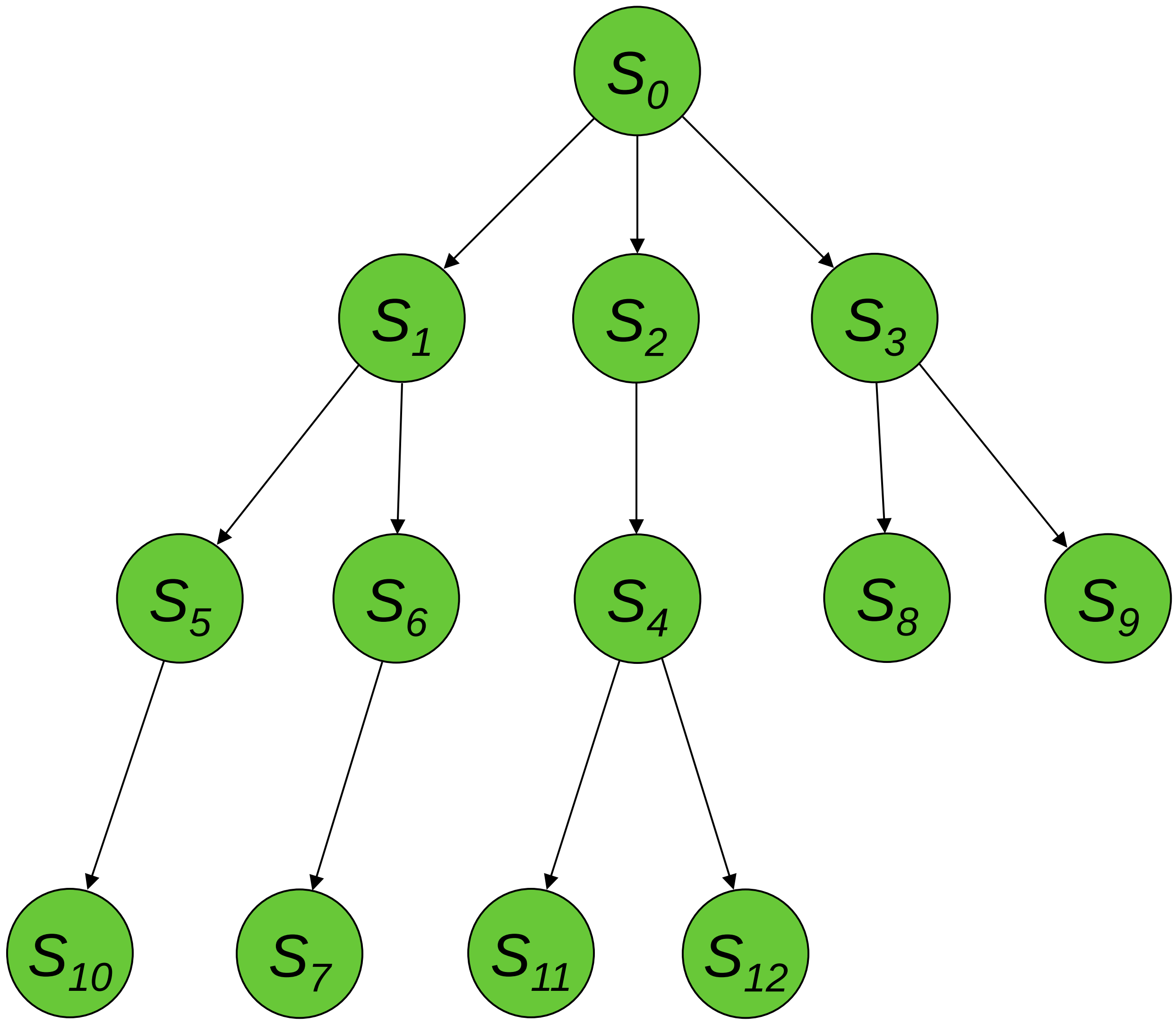
1. A representation for our states: where is the truck, where are the packages, how much petrol is left
2. The initial state of the world
3. A goal state recogniser
4. The NextStates(State) function

You are now ready to apply a search algorithm…

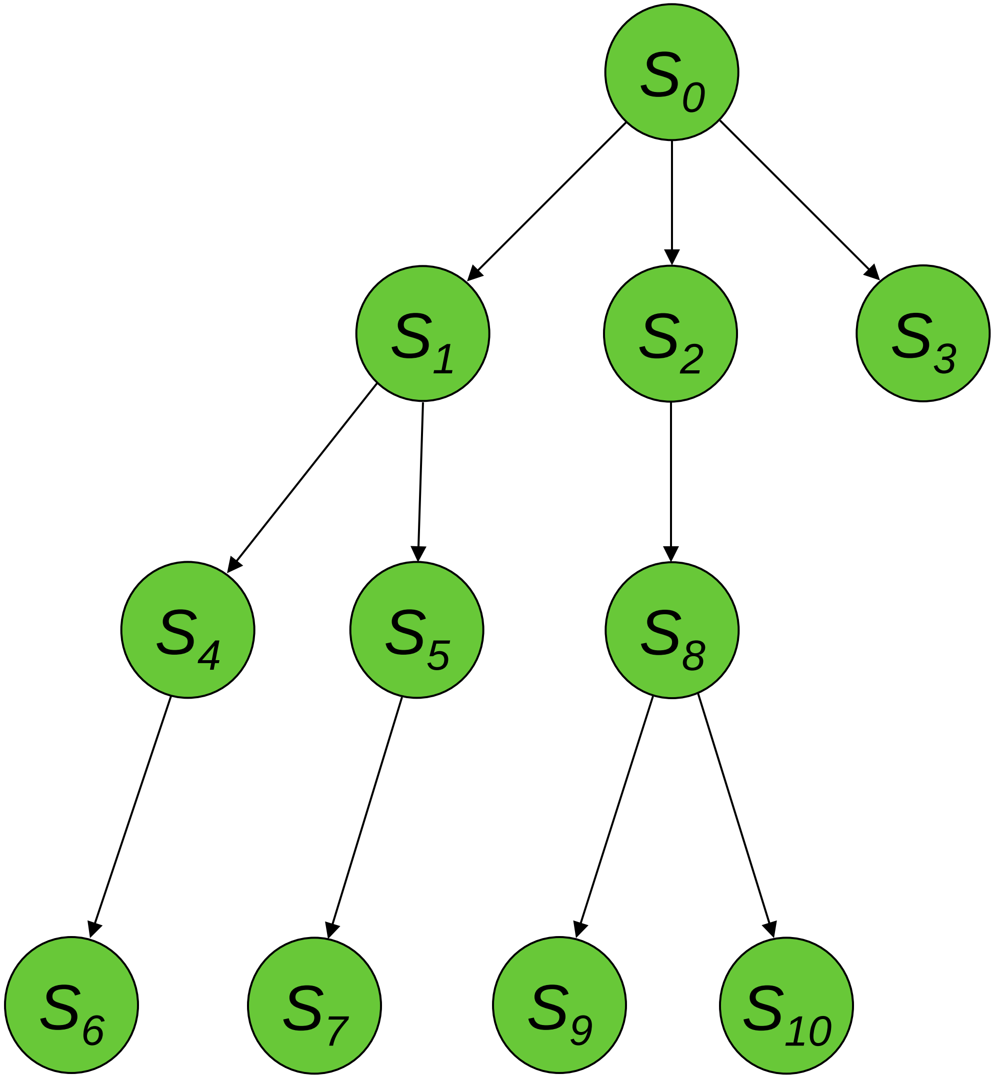
Search Spaces

- The search space of a problem is implicit in its formulation
  - You search the space of your representations
- We generate the space dynamically during search (including loops, dead ends, branches)
- Operators are move generators
- We can represent the search space with trees
- Each node in the tree is a state
- When we call NextStates(S_0) → [S_1,S_2,S_3], then we say we have expanded S_0

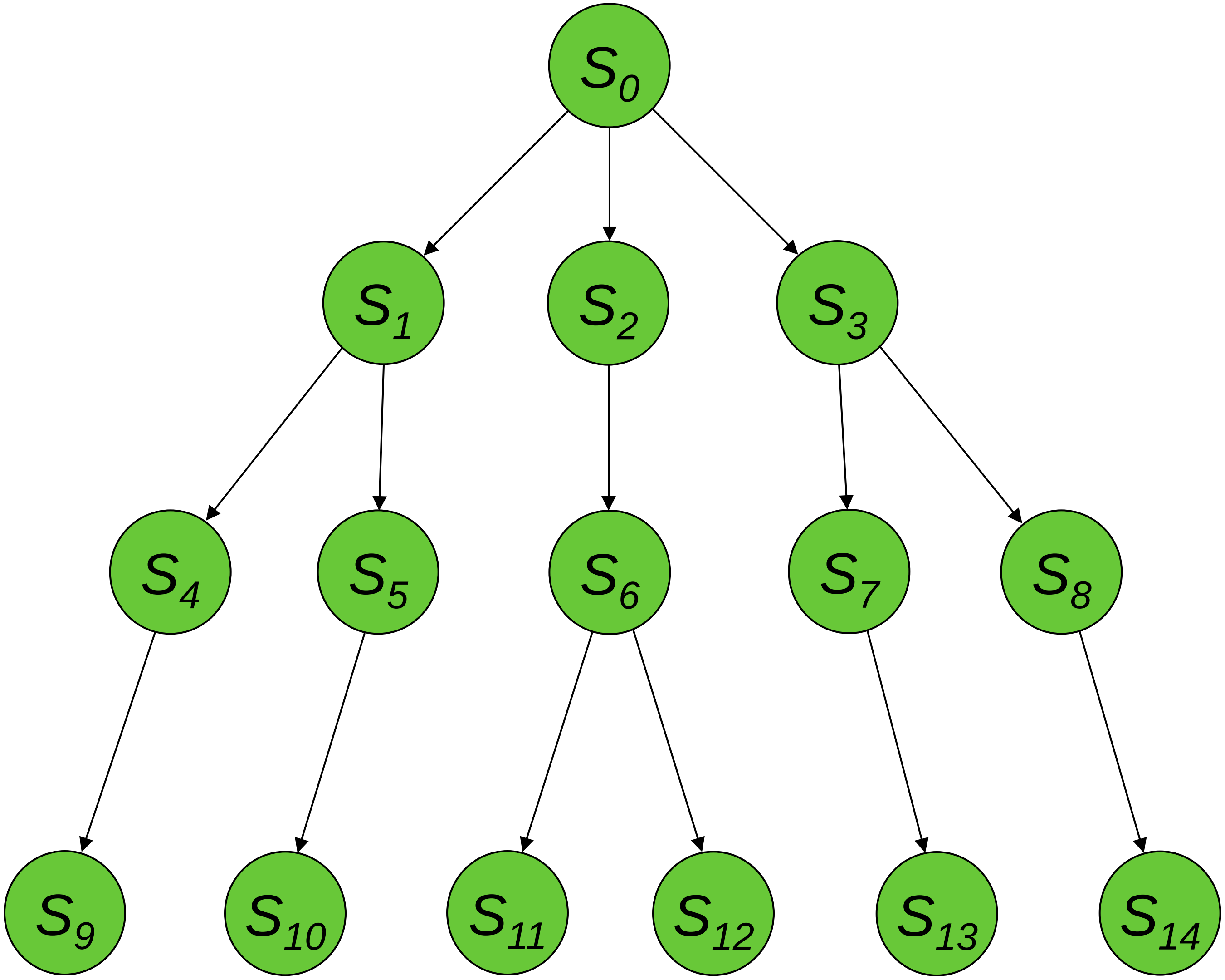
Expanding Nodes in the Search Space



Depth-First Search



Breadth-First Search



Searching Using an Agenda

- When we expand a node we get multiple new nodes to expand, but we can only expand one at a time
- We keep track of the nodes still to be expanded using a data structure called an agenda
- When it is time to expand a new node, we choose the first node from the agenda
- As new states are discovered, we add them to the agenda somehow
- We can get different styles of search by altering how the states get added

Depth-First Search

- To get depth-first search, add the new nodes onto the start of the agenda (treat the agenda as a stack):

let Agenda = [S_0]
while Agenda ≠ [] do
    let Current = remove-first(Agenda)
    if Goal(Current) then return(“Found it!”)
    let Next = NextStates(Current)
    let Agenda = Next+Agenda

Breadth-First Search

- To get breadth-first search, add the new nodes onto the end of the agenda (treat the agenda as a queue):

let Agenda = [S_0]
while Agenda ≠ [] do
    let Current = remove-first(Agenda)
    if Goal(Current) then return("Found it!")
    let Next = NextStates(Current)
    let Agenda = Agenda+Next

Properties of Depth-First Search

- Depth-first can often get to the answer quickly
- The agenda stays short: $O(d)$ for memory, where d is the depth of the tree
- The time taken to find the solution is $O(d)$ in the best case and $O(b^d)$ in the worst case (where b is the average branching factor)
- But if there are loops in the search space, it can get into an infinite loop
- It isn't guaranteed give the shortest solution

Properties of Breadth-First Search

- Breadth-first can often take a long time to get to the answer
- The agenda can get very big: $O(b^d)$ for memory, giving exponential space consumption
- Also exponential time complexity: $O(b^d)$ nodes will be expanded
- But it isn't bothered by loops in the search space
- And it always gives the shortest solution, in terms of the number of steps in the plan