Automated Planning and Scheduling

Lecture 3

Tomáš Balyo, Dominik Schreiber | November 3, 2019
State Space Search Algorithms

- State Space Properties
  - Combinatorial Explosion
- Forward Search
  - Blind: DFS, BFS, Dijkstra’s algorithm
  - Heuristic: BestFistSearch, A*, Enforced Hill Climbing
- Backwards Search
- Bi-Directional Search
State Space Graph

The trucking example again:

Consider the FDR
Let us represent each world state as XYZ where
- $X \in \{A, B, C\}$ is the truck location
- $Y, Z \in \{A, B, C, T\}$ are the locations of the packages

Initial state: AAB, goal: *CC
All possible states: $3 \times 4 \times 4 = 48$
State Space Graph

Planning: find a path from AAB to CCC
State Space Graph

- So planning is just path-finding :-)
- We can use standard path search algorithms
  - Breadth-first search
  - Depth-first search
  - Dijkstra’s algorithm
  - ...

So planning is just path-finding :-)

We can use standard path search algorithms

- Breadth-first search
- Depth-first search
- Dijkstra’s algorithm
- ...

BUT...
Example – All Paths in a grid

https://youtu.be/Q4gTV4r0zRs
Example – Rubik’s Cube

3x3x3 Cube

- $4.3 \times 10^{19}$ combinations
- A standard cube has a side length 5.7 cm, if you assemble $4.3 \times 10^{19}$ standard cubes into a huge cube it would have a side of 200 km.

4x4x4 Cube

- $7 \times 10^{45}$ combinations
- The huge cube side would be 0.1 light years.

5x5x5 Cube

- $2 \times 8 \times 10^{74}$ combinations
- The huge cube side would be 10 million light years (our own galaxy + 70 nearest galaxies would fit in that cube).
Example – Rubik’s Cube

### 3x3x3 Cube
- $4.3 \times 10^{19}$ combinations
- A standard cube has a side length 5.7 cm, if you assemble $4.3 \times 10^{19}$ standard cubes into a huge cube it would have a side of 200 Km.

### 4x4x4 Cube
- $7.4 \times 10^{45}$ combinations
- The huge cube side would be 0.1 Light years
## Example – Rubik’s Cube

### 3x3x3 Cube
- \(4.3 \times 10^{19}\) combinations
- A standard cube has a side length 5.7 cm, if you assemble \(4.3 \times 10^{19}\) standard cubes into a huge cube it would have a side of 200 Km.

### 4x4x4 Cube
- \(7.4 \times 10^{45}\) combinations
- The huge cube side would be 0.1 Light years

### 5x5x5 Cube
- \(2.8 \times 10^{74}\) combinations
- The huge cube side would be 10 million Light years (our own galaxy + 70 nearest galaxies would fit in that cube).
So planning is just path-finding :-) 
We can use standard path search algorithms
  - Breadth-first search
  - Depth-first search
  - Dijkstra’s algorithm
  - ...

BUT

- The graph is astronomically huge
- Does not fit in the memory
- We will generate it on the go
Depth-First Search

function DFS(State s) {
    if (goal(s)) return True
    foreach (Action a in applicable(s)) {
        if (DFS(apply(s,a)) return True
    }
    return False
}

The function DFS(init) will return True if a plan exists
Demo at https://www.movingai.com/SAS/EXP/
DFS Properties

Advantages:
- linear time complexity (in found plan size)
- can find long plans fast (if we are lucky)
- easy to implement

Disadvantages:
- May get into loops
  - we can remember and then avoid visited states, increases memory complexity (linear in visited states).
  - Still exponential time complexity
- May find very inefficient plans
function BFS(State init) {
    Queue<State> sq = {init}
    while (sq not empty) {
        s = sq.dequeue()
        if (goal(s)) return True
        foreach (Action a in applicable(s)) {
            sq.enqueue(apply(s,a))
        }
    }
    return False
}

The function BFS(init) will return True if a plan exists
BFS Properties

Advantages:
- finds optimal plans (in number of actions)
- sound and complete

Disadvantages:
- Exponential time and space complexity (in plan length)
- No amount of luck will help us if the shortest plan is long
Dijkstra’s Algorithm

If we action cost and want optimal plans we can use Dijkstra’s algorithm:

```java
function Dijkstra(State init) {
    PriorityQueue<State, int> sq
    sq.add(init, 0)
    while (sq not empty) {
        s = sq.getMin() // smallest cost to init
        if (goal(s)) return Plan
        foreach (Action a in applicable(s)) {
            sq.add(apply(s,a), cost(s) + cost(a))
        }
    }
    return False
}
```
Dijkstra Properties

Advantages:
- finds optimal plans
- sound and complete

Disadvantages:
- Exponential time and space complexity (in plan length)

Demo: https://www.movingai.com/SAS/ASG/
Heuristics

- The previous algorithms are unusable for planning in most cases
- They are too uninformed (blind)
Heuristics

- The previous algorithms are unusable for planning in most cases
- They are too uninformed (blind)

Heuristic

A **heuristic** function $f : S \rightarrow \mathbb{R}$ estimates the cost of a plan from state $s \in S$ to the nearest goal state.

Admissible Heuristic

Let $f$ be a heuristic and $o(s)$ be the cost of the optimal plan from $s$ to the nearest goal state. We say that $f$ is an **Admissible heuristic** if

$$\forall s \in S : f(s) \leq o(s)$$
function GBFS(State s) {
    if (goal(s)) return True
    if (visited(s)) return False
    visited(s):=true
    A = actions applicable in state s
    sort(A) // according to h(apply(s,a))
    foreach (Action a in A) {
        if (GBFS(apply(s,a)) return True
    }
    visited(s):=false
    return False
}
Greedy Best First Search Properties

- GBFS chooses the action with the lowest heuristics value first
- Very fast on simple domains given a good heuristic
- May produce very inefficient plans
- Does not work well on complex domains
- Uses a lot of memory to store visited states
How would you implement GBFS?

- How to implement testing visited states?
How would you implement GBFS?

- How to implement testing visited states?
  - Hash Tables
How would you implement GBFS?

- How to implement testing visited states?
  - Hash Tables
How would you implement GBFS?

- How to implement testing visited states?
  - Hash Tables
  - Lossy hash table
How would you implement GBFS?

- How to implement testing visited states?
  - Hash Tables
  - Lossy hash table
- How to implement ”get applicable actions”?
How would you implement GBFS?

- How to implement testing visited states?
  - Hash Tables
  - Lossy hash table

- How to implement "get applicable actions"?
  - Index all action based on preconditions
How would you implement GBFS?

- How to implement testing visited states?
  - Hash Tables
  - Lossy hash table

- How to implement ”get applicable actions”?
  - Index all action based on preconditions
  - Calculate from previous set of applicable actions and the delta of previous and current state.
    - remove actions that are not applicable anymore
    - add actions that became applicable based on delta
function BestFS(State init) {
    PriorityQueue<State, int> sq
    sq.add(init, heuristic(s))
    while (sq not empty) {
        s = sq.getMin() // smallest heuristic value
        if (goal(s)) return Plan
        foreach (Action a in applicable(s)) {
            sq.add(apply(s,a), heuristic(apply(s,a))
        }
    }
    return False
}
Best First Search Properties

- Similar to Breadth First Search and Dijkstra's algorithm
- Keeps a queue of open nodes and expands the one with best heuristic value
- Needs a lot of memory if the heuristic has plateaus
- Does not find optimal plans
function A*(State init) {
    PriorityQueue<State, int> sq
    sq.add(init, 0 + h(s))
    while (sq not empty) {
        s = sq.getMin()    // smallest value
        if (goal(s)) return Plan
        foreach (Action a in applicable(s)) {
            sq.add(apply(s,a), cost(s) + cost(a) + h(apply(s,a))
        }
    }
    return False
}
A* Properties

- A combination of Dijkstra’s algorithm and BestFS
- Expands a node with minimal \( f(s) = g(s) + h(s) \) value
  - \( g(s) \) the distance to initial state (like in Dijkstra)
  - \( h(s) \) the heuristic value
- If the heuristic is admissible then A* is optimal
- A* is optimally efficient (expands the minimal amount nodes necessary to find the optimal solution)
- A* is the most popular ”A.I. Algorithm” ever
Enforced Hill Climbing

- EHC is a combination of Greedy Best First Search and Best First Search.
- It runs like GBFS until a plateau is reached (all applicable actions lead to states with same heuristic value).
- Then it switches to Best First Search (breadth first style) until a better state is reached.
- EHC search usually consists of prolonged periods of exhaustive search (BFS) connected by relatively short periods of heuristic descent.
- Invented by Joerg Hoffmann in 2005 for the planner FF.
Backwards Search
Backwards Search

Basic Idea:

- We search from goal towards the initial state
- Relevant actions for a set of atoms = actions with relevant effects
- Starting with the goal conditions we choose one of the relevant actions and “un-apply” it to reach the next state
- Repeat until the initial state is reached

How to “un-apply” an action $a$ in a partial state $s$ (in PDR)?

- remove effects of $a$ from $s$
- add preconditions of $a$ to $s$
Backward Search for Trucks

The trucking example again:

Let us represent each world state as $XYZ$ where
- $X \in \{A, B, C\}$ is the truck location
- $Y, Z \in \{A, B, C, T\}$ are the locations of the packages

Initial state: AAB, goal: *CC
Find a path from *CC to AAB
Backward Search

- Still requires search because many actions may be relevant
- For some domains the branching factor may be smaller when searching backwards
- Same search algorithm as for forward search may be used
- Similar heuristics can be designed
- It is possible to reformulate a problem such that forward search simulates backwards search
Bi-Directional Search

Basic Idea

- Search simultaneously from both initial state and goal and meet in the middle
- Maintain 2 collections of open nodes
- Use heuristics to guide the search
- Theoretically faster than both backwards and forwards search
  - let the $b$ the branching factor and $d$ the plan length
  - forward and backward search (BFS) would expand $O(b^d)$ nodes
  - Bi-Directional Search would only expand $O(b^{d/2})$

Demo: https://www.movingai.com/SAS/NBS/
The End

Next Week: Heuristics for state space search