**Artificial Intelligence**

**CoMP30024**

Shromann majumder

2021

Intelligent Agent

Agent Model

* Percept

*What the agent sees*

* Actions

*What the agent does*

* Environment

*The context the agent is in*

* Performance

Measure: *how good/bad is the* ***action*** *for the agent.*

Ex. *autonomous car* (depends on the objective of the problem)

1. Percept: video, accelerometer…
2. Action: accelerate, steer…
3. Environment: road, traffic, people…
4. Performance: fastest route, safety…

**Agents as functions**

* Agent is a function from **percept** sequences to **action**.
* Ideal ***Rational*** Agents maximise **performance**
  + ***Rational*** is:
    - **NOT** Omniscient (doesn’t know the outcome)
    - **NOT** Clairvoyant (doesn’t know next step)
    - **NOT** Successful (not always successful, as a probability assigned to each step)

Types of Agents

1. simple reflex

environment: **fully-**observable

Diagram

Description automatically generated

actions in the moment, no memory

1. model-based

environment: **partially-**observable

Diagram

Description automatically generated

Internally models environment with storing past states, then back to ***simple reflex***

1. goal-based

models how world looks like w.r.t action and act w.r.t goal.

Diagram

Description automatically generated

1. utility-based

prioritise goals that satisfy the overall objective

Diagram

Description automatically generated

Environment Types

1. Observable

We know everything we need to know

1. Deterministic

Current state impacts the next state

1. Episodic

Only current or recent state(s) are relevant

1. Static

Environment doesn’t change

1. Discrete

Finite number of possible states

Ex. Solitaire, Backgammon, Internet Shopping, Taxi

Table

Description automatically generated

Problem Solving and Search

* Special type of **Goal-Based** Agent

Problem-Solving Agents

* form of ***Offline*** problem solving (static environment)
* class of agent that follows the algorithm:

function **Simple-Problem-Solving-Agent**(**p**)

**state** = update-state(**state**, **p**)

if **s** is empty then

**g** = Formulate-Goal(**state**)

**problem** = Formulate-Problem(**state**, **g**)

**s** = Search(**Problem**)

**action** = Recommendation(**s**, **state**)

**s** = Remainder(**s**, **state**)

return **action**

|  |  |
| --- | --- |
| **s** | action sequence |
| **State** | current world state |
| **g** | goal state |
| **problem** | formulated problems |

1. Formulate Goal

This is in the form of **state**. Moreover, it is the **final** **& desired** state.

1. Formulate Problem
2. States

A particular representation of the world that the agent can understand.

1. Operators

Actions that an agent can take to go from one state to another.

1. Find Solution

**Sequence** of *states/operators* that the agent makes to reach **final** **& desired** state.

Ex. Romania

Goal: Get to Bucharest

Problem: States: Cities, Operators: Driving

Solution: Find Path

Single-State Problem Formulation

1. Initial state

Where do we begin???

1. Actions

How do I go from one state to another???

1. Goal test
2. Explicit

X = “Bucharest”

1. Implicit

In checkmate

What state do I stop???

1. Path cost

How expensive/cheap is getting to a particular state is??

Selecting State Space

* Restate the real-world problem in an **abstract** space to allow for computations.
* Kind of like representing a **whole city** as a *simple* **dot**.

Search Algorithms

* *Offline* + Simulated Exploration of **State Space**
* **Node** is a state that is encoded in the **state space**

function General-Search(problem, strategy)

Loop do

If nothing to expand:

return Failure

node = leaf-node(strategy)

if node is goal-state:

return Success

else:

search-tree.append(expand(node))

Implementation of Search Algorithm

* Implementing a search algorithm is essentially defining the strategy from the above pseudocode to a *queuing-fn*.
* This is the **order of node expansion**.

function General-Search(problem, *queuing-fn*)

nodes = Make-Queue(Make-Node(Initial-State[problem]))

Loop do

If nothing to expand:

return Failure

node = POP(nodes)

if node is goal-state:

return Success

else:

nodes = *queuing-fn*(nodes, Expand(node, Operations(Problem)))

Search Strategies

* As said above, A search strategy = order of node expansion = *queuing-fn*.

Evaluation

1. Completeness

Does it always find a solution??

1. Time-Complexity

Total number of nodes expanded

1. Space-Complexity

Maximum number of nodes in memory

Time and Space are measured with

* + - * *b* (maximum branching factor)
      * *d* (depth of the cheapest solution)
      * *m* (maximum depth of the state space, may be infinity)

1. Optimality

Does it always find the best/optimal solution??

Uninformed Search Strategies

* Only use information available in the problem definition

1. BFS

*queuing-fn* = put successors at the **end** of the queue

|  |  |
| --- | --- |
| Complete | Yes (b is finite) |
| Time |  |
| Space |  |
| Optimal | Yes |

1. Uniform-Cost Search

*queuing-fn* = order of increasing path cost

|  |  |
| --- | --- |
| Complete | Yes (step cost >e) |
| Time |  |
| Space |  |
| Optimal | Yes |

1. Depth-First Search

*queuing-fn* = put successors at the **end** of the queue

|  |  |
| --- | --- |
| Complete | No (infinite loops in infinite space) |
| Time |  |
| Space |  |
| Optimal | No |

**m**, maximum depth of state space

1. Depth-Limited Search

*queuing-fn* = put successors at the **end** of the queue

DFS with Depth-Limit: ***L***

|  |  |
| --- | --- |
| Complete | Yes |
| Time |  |
| Space |  |
| Optimal | No |

1. Iterative Deepening Search

Do DFS at depth, but increase depth if goal state not in-depth ***L***

|  |  |
| --- | --- |
| Complete | Yes |
| Time |  |
| Space |  |
| Optimal | Yes |

1. Bidirectional Search
   * Search from **start** *to* **goal & goal** *to* **start**
   * Stop when both paths intersect.

|  |  |
| --- | --- |
| Complete | Yes |
| Time |  |
| Space |  |
| Optimal | Yes (if using correct strategy - BFS) |

Informed Search Strategies

Best-First Search

* Use **Evaluation** function, which *estimates* desirability.
* Expand most desirable unexpanded node.

Implementation

*Queueing-Fn*: insert successors in **decreasing** order of **desirability**.

*Special Types of Best-First Search:*

* Greedy Search
* Evaluation Function: estimate cost from **n** to **goal** (Euclidean-Distance)

Properties

1. Complete: **No**, may get caught up in a loop. **Yes**, if finite state-space
2. Time: – but a good heuristic can improve it
3. Space: – all nodes in memory
4. Optimal: **No**

* A\* Search
* Evaluation Function: f(n) = g(n) + h(n)
  + g(n) = cost so far to reach n
  + f(n) = cost from n to goal
  + h(n) = total cost from start to goal
* A\* uses an *admissible* heuristic: h(n) < h\*(n), where h\*(n) is the **true** cost from n
* Thus, A\* Search is **Optimal**

Properties

1. Complete: **Yes**, unless infinitely many nodes
2. Time: **Exponential**
3. Space: **Keeps all nodes in memory**
4. Optimal: **Yes**

Different heuristics can lead to different search algorithms, as shown belowText, letter

Description automatically generated

Heuristics

Common Heuristics

* Manhattan distance (L1): On a square grid that allows 4 directions of movement.
* Diagonal distance (L∞): On a square grid that allows 8 directions of movement.
* Euclidean distance (L2): On a square grid that allows any direction of movement.

Admissible Heuristics

* *admissible* heuristic: h(n) < h\*(n), where h\*(n) is the **true** cost from n is **Optimal.**

Dominance

* If h2(n) > h1(n) for all n (both admissible), then h2 **dominates** h1, thus a **better** search.

Relaxed Problem

* Admissible heuristics can be derived from the exact solution cost of a relaxed version of the problem

Iterative improvement algorithms

* In many optimization problems, path is irrelevant; the goal state itself is the solution
* Then state space = set of “complete” configurations
  + Find optimal configuration, e.g., Travelling Salesperson Problem or, find configuration satisfying constraints, e.g., n-queens
* In such cases, can use **iterative improvement algorithms**.
  + keep a single “current” state, try to improve it
* Constant space, suitable for online as well as offline search

Ex. Travelling Salesperson Problem, n-Queens

Hill-Climbing (gradient descent/ascent)

* Like climbing Everest in thick fog with amnesia
* Pseudocode:

function Hill-Climbing(problem)

*current* = make-node(Initial-State[problem])

loop do

*next* = a highest-valued successor of *current*

if Value[*next*] < Value[*current*]

return *current*

*current* = *next*

Chart, line chart

Description automatically generatedProblem

* depending on initial state, can get stuck on local maxima

Questions

1. What do the different types of environments really mean?
2. Why do the Uninformed Searches have their evaluations?