CS 581 Notes

0.1 Agent

An agent is just something that acts (from the Latin agere, to do).

Of course, we would prefer "acting" to be:

- autonomous
- situated in some environment (that could be really complex)
- adaptive
- create and goal-oriented

0.2 Rational Agent

A rational agent is one that acts to achieve the best outcome, or when there is uncertainty, the best expected outcome[1].

0.3 AI: Constructing Agents

You can say that: AI is focused on the study and construction of agents that do the right thing.

0.4 Percepts and Percept Sequences

Percept – content/information that agent's sensors are perceiving / capturing currently

Percept Sequence – a complete history of everything that agent has ever perceived

- any practical issues that you can see here?
- what can a percept sequence be used for?

0.5 Agent Function/Program

Specifying an action choice for every possible percept sequence would define an agent

- Action <-> percept sequence mapping IS the agent function.
- Agent function describes agent behavior.

[1] no worries, we will make it a little less vague soon

- Agent function is an abstract concept.
- Agent program implements agent function.

0.6 Actions Have Consequences

An agent can act upon its environment, but how do we know if the end result is "right"? After all, actions have consequences: either good or bad. Recall that agent actions change environment state! If state changes are desirable, and agent performs well. Performance measure evaluates state changes.

0.7 Rationality

Rational decisions at the moment depend on:

- The performance measure that defines success criteria
- The agent's prior knowledge of the environment
- The actions that the agent can perform
- The agent's percept sequence so far

0.8 Rational Agent

For each possible percept sequence, a rational agent should select an action that is expected to maximize

0.9 Rationality in Reality

- An omniscient agent will ALWAYS know the final outcome of its action. Impossible in reality. That would be perfection.
- Rationality maximizes what is EXPECTED to happen.
- Perfection maximizes what WILL happen.
- Performance can be improved by information gathering and learning.

0.10 Designing the Agent for the Task

0.10.1 Analyze the Problem

Task Environment — PEAS

In order to start the agent design process, we need to specify/define:

- The Performance measure
- The Environment in which the agent will operate
- The Actuators that the agent will use to affect the environment
- The Sensors

Task Environment Properties

Key dimensions by which task environments can be categorized:

- Fully vs partially observable (can be unobservable too)
- Single agent vs multi-agent
  - multi-agent: competitive vs. co-operative
- Deterministic vs. non-deterministic (stochastic)
- Episodic vs. sequential
  - Sequential requires planning ahead
- Static vs. dynamic
- Discrete vs. continuous
- Known vs. unknown (to the agent)

0.10.2 Select Agent Architecture

$$\textbf{Agent} = \textbf{Architecture} + \textbf{Program}$$

Typical Agent Architectures

- Simple reflex agent.
- Model-based reflex agent.
- Goal-based reflex agent.
- Utility-based reflex agent.

0.10.3 Select Internal Representations

0.11 Typical Agent Architectures

- Simple reflex agent: uses condition-action rules
- Model-based reflex agent: keeps track of the unobserved parts of the environment by maintaining internal state:
  - "how the world works": state transition model
  - how percepts and environment is related: sensor model
- Goal-based reflex agent: maintains the models of the world and goals to select decisions (that lead to goal)
- Utility-based reflex agent: maintains the model of the world and utility function to select PREFERRED decisions (that lead to the best expected utility: avg (EU * p))

0.12 State and Transition Representations

- Atomic: state representation has NO internal structure
- Factored: state representation includes fixed attributes (which can have values)
- Structured: state representation includes objects and their relationships

0.13 Problem-Solving / Planning Agent

- Context / Problem:
  - correct action is NOT immediately obvious
  - a plan (a sequence of actions leading to a goal) may be necessary
- Solution / Agent:
  - come up with a computational process that will search for that plan
- Planning Agent:
  - uses factored or structured representations of states
  - uses searching algorithms

0.14 Defining Search Problem

- Define a set of possible states: State Space
- Specify Initial State
- Specify Goal State(s) (there can be multiple)
- Define a FINITE set of possible Actions for EACH state in the State Space
- Come up with a TRANSITION model which describes what each action does
- Specify the Action COST Function (a function that gives the cost of applying action a to state s)

0.15 Measuring Searching Performance

Search algorithms can be evaluated in four ways:

- Completeness: Is the algorithm guaranteed to find a solution when there is one, and to correctly report failure when there is not?
- Cost optimality: Does it find a solution with the lowest path cost of all solutions?
- Time complexity: How long goes it take to find a solution? (in seconds, actions, states, etc.)
- Space complexity: How much memory is needed to perform the search?

0.16 Informed Search and Heuristics

Informed search relies on domain-specific knowledge / hints that help locate the goal state.

$$h(n) = h(\text{State } n)$$
$$h(n) = n(\text{relevant information about State } n)$$

0.17 Romanian Roadtrip: Heuristics h(n)

For this particular problem, the heuristic function $h(n)$ is defined by a straight line (Euclidean) distance between two states (cities). As the crow flies in other words.

0.18 A* Algorithm: Evaluations Function

Calculate/obtain:

$$f(n) = g(\text{State}_n) + h(\text{State}_n)$$

0.19 A* Evaluation Function

$$f(n) = g(\text{State}_n) + h(\text{State}_n)$$

where:

- $g(n)$ – initial node

0.20 Admissible Heuristic: Proof

An admissible heuristics $h()$ is guaranteed to give you the optimal solution. Why? Proof by contradiction:

- Say: the algorithm returned a suboptimal path ($C > C*$)
- So: there exists a node n on $C*$ not expanded on C:

If so:

$$\begin{aligned} &f(n) > C* \\ &f(n) = g(n) + h(n) && \text{(by definition)} \\ &f(n) = g*(n) + h(n) && \text{(because } n \text{ is on } C*) \\ &f(n) \leq g*(n) + h*(n) && \text{(if } h(n) \text{ admissible: } h(n) \leq h*(n) \end{aligned}$$

0.21 What Made A* Work Well?

- Straight-line heuristics is consistent: its estimate is getting better and better as we get closer to the goal
- Every consistent heuristics is admissible heuristics, but not the other way around

But that would mean that:

$$f(n) \leq C*$$

0.22 A*: Search Contours

How does A* "direct" the search progress?

0.23 Dominating Heuristics

We can have more than one available heuristics. For example $h_1(n)$ and $h_2(n)$. $h_2(n)$ dominates $h_1(n)$ iff[2] $h_2(n) > h_1(n)$ for every n.
If you have multiple admissible heuristics where none dominates the other:

$$\text{Let } h(n) = \max\left(h_1(n), h_2(n), \ldots, h_m(n)\right)$$

[2] if and only if

0.24 Domination → Efficiency: Why?

With

$$f(n) < C* \text{ and } f(n) = g(\text{State}_n) + h(\text{State}_n),$$

we get

0.25 Domination → Efficiency: But?

If $h_2(n)$ dominates $h_1(n)$, should you always use $h_2(n)$? Generally yes, but $h_2(n)$ vs $h_1(n)$ heuristic *computation time* may be a deciding factor here.

0.26 Heuristic and Search Performance

- Consider an 8-puzzle game and two admissible heuristics:
  - $h_1(n)$ – number of misplaced tiles (not counting blank)
  - $h_2(n)$ – Manhattan distance

0.27 $h()$ Quality: Effective Branching

0.28 Can We Make A* Even Faster? (Sometimes at a cost!)

0.29 Weighted A* Evaluation Function

$$f(n) = g(\text{State}_n) + W * h(\text{State}_n)$$

where:

- $g(n)$ – initial node to node n path cost
- $h(n)$ – estimated cost of the best path that continues from node n to a goal node
- $W > 1$

Here, weight W makes $h(\text{State}_n)$ (perhaps only "sometimes") inadmissible. It becomes potentially more accurate = less expansions!
[autoplay,loop,width=] 10file

0.30 Perfect Information

0.31 Two Player Games: Env Assumptions

0.32 Defining Zero Sum Game Problem

- Defining a set of possible states: State Space
- Specify how will you track Whose Move / Turn it is
- Specify Initial State
- Specify Goal State(s) (there can by multiple)
- Define a FINITE set of possible Actions (legal moves) for EACH state in the State Space
- Come up with a Transition Model which describes what each action does
- Come up with a Terminal Test that verifies if the game is over
- Specify the Utility (Payoff/Object) Function: a function that defines the final numerical value to player p when the game ends in the terminal state $s4$

0.33 MinMax Algorithm: The Idea

I don't know what move my opponent will choose, but I am going to ASSUME that it is going to be the best/optimal option for them.

α the value of the best (highest value)

β

0.34 Constraint Satisfaction Problem

The goal is to find an assignment (variable = value):

$$X_1 = v_1, \ldots, X_n = v_n$$

- If NO constraints violated: consistent assignment
- If ALL variables have a value: complete assignment
- If SOME variables have NO value: partial assignment
- SOLUTION: consistent and complete assignment
- PARTIAL SOLUTION: consistent and partial assignment

Chapter 1

Local Search Algorithms

1.1 "Hill Climbing" (Greedy Local)

Assumption: We don't go to a repeated state

Do we always need to care about the path to the goal?

1.2 Informed Search: the Idea

When traversing the search tree, use domain knowledge / heuristics to avoid search paths (moves/actions) that are likely to be fruitless

1.3 Informed Search and Heuristics

Informed search relies on domain-specific knowledge/hints that help locate the goal state.

1.4 Hard Problems

- Many important problems are provably not solvable in polynomial time (NP and harder)
- Results based on the worst-case analysis
- In practice: instances are often easier
- Approximate methods can often obtain good solutions

1.5 Local Search

- Moves between configurations by performing local moves
- Works with complete assignments of the variables

- Optimization problems:
  - Start from a suboptimal configuration
  - Move towards better solutions
- Satisfaction problems:
  - Start from an infeasible configuration
  - Move towards feasibility
- NO guarantees
- Can work great in practice!

1.6 Local Search Algorithms

If the path to the goal does not matter, we might consider a different class of algorithms. Local Search Algorithms

- do not worry about paths at all.
- Local Search

1.6.1 Selecting Neighbor

- How to select the neighbor?
  - exploring the whole or part of the neighborhood
- Best neighbor
  - Select "the" best neighbor in the neighborhood
- First neighbor
  - Select the first "legal" neighbor
  - Avoid scanning the entire neighborhood
- Multi-stage selection
  - Select one "part" of neighborhood and then
  - select from the remaining "part" of the neighborhood
- Hill-climbing search
  - Gradient descent in continuous state spaces
  - Can use e.g. Newton's method to find roots

- Simulated annealing search
- Tabu search
- Local beam search
- Evolutionary/genetic algorithms

Although local search algorithms are not systematic, they have two key advantages:

- they use very little memory – usually a constant amount; and
- the can often find reasonable

Local search algorithms are useful for search pure optimization problems, in which the aim is to find the best state according to the objective function

1.7 Simulated Annealing

1.7.1 What Is It?

In metallurgy, annealing is the process used to temper or harden metals and glass by heating them to a high temperature T and then gradually cooling them, thus allowing the material to coalesce into a low-energy E crysalline state (less or no defects).
Key Idea:

- Use Metropolis algorithm but adjust the temperature dynamically
- Start with a high temperature (random moves)
- Decrease the temperature
- When the temperature is low, becomes a local search

1.8 Metropolis Heuristics

1.8.1 Basic Idea

- Accept a move if it improves the objective value
- Accept "bad moves" as well with some probability
- The probability depends on how "bad" the move is
- Inspired by statistical physics

1.8.2 How to choose the probability?

- t is a scaling parameter (called temperature)
- Δ is the difference $f(n) - f(s)$

1.8.3 Fixed T

- What happens for a large T?
  - Probability of accepting a degrading move is large
- What happends for a small T?
  - Probability of accepting a degrading move is small

Algorithm 1.1 Simulated Annealing Pseudocode

```
1: function SIMULATED-ANNEALING(problem, schedule) returns a solution state
2:     current ← problem.INITIAL
3:     t ← 1
4:     while True do
5:         T ← SCHEDULE(t)
6:         if T == 0 then return current
7:         end if
8:         next ← a randomly selected successor of current
9:         ΔE ← VALUE(current) − VALUE(next)
10:        if ΔE > 0 then
11:            current ← next
12:        else
13:            current ← next only with probability e^{−ΔE/T}
14:        end if
15:    end while
16: end function
```

1.8.4 Temperature/Cooling Schedule

Idea: start with

1.8.5 Summary

- Converges to a global optimum
  - connected neighborhood
  - slow cooling schedule
    - *slower than the exhaustive search*

- In practice
  - can give excellent results
  - need to tune a temperature schedule
  - default choice: $t_{k+1} = \alpha t_k$
- Additional tools
  - restarts and reheats

1.8.6 Applications

- Basic Problems
  - Traveling salesman
  - Graph partitioning
  - Matching problems
  - Graph coloring
  - Scheduling
- Engineering
  -

1.9 Heuristics and Metaheuristics

- Heuristics
  - how to choose the next neighbor?
  - use local information (state and its neighborhood)
  - direct the search towards a local min/maximum
- Metaheuristics
  - how to escape local minima?
  - direct the search towards a global min/maximum
  - typically include some memory or learning