State Space Search

Overview

- Problem-solving as search
- How to formulate an AI problem as search.
- Uninformed search methods

What is search?

Environmental factors needed

- **Static** — The world does not change on its own, and our actions don't change it.
- **Discrete** — A finite number of individual states exist rather than a continuous space of options.
- **Observable** — States can be determined by observations.
- **Deterministic** — Action have certain outcomes.

- The **environment** is all the information about the world that remains constant while we are solving the problem.
- A **state** is a set of properties that define the current conditions of the world our agent is in.
  - Think of this as a *snapshot* of the world at a given point in time.
  - The entire set of possible states is called the **state space**.
- The **initial state** is the state the agent begins in.
- A **goal state** is a state where the agent may end the search.
- An agent may take different **actions** that will lead the agent to new states.

Formulating problems as search

- *Canonical problem*: route-finding
  - Route-finding with traveling salesperson problem.
- Sliding block puzzle (almost any kind of game or puzzle can be formulated this way).
- Water jug problem.

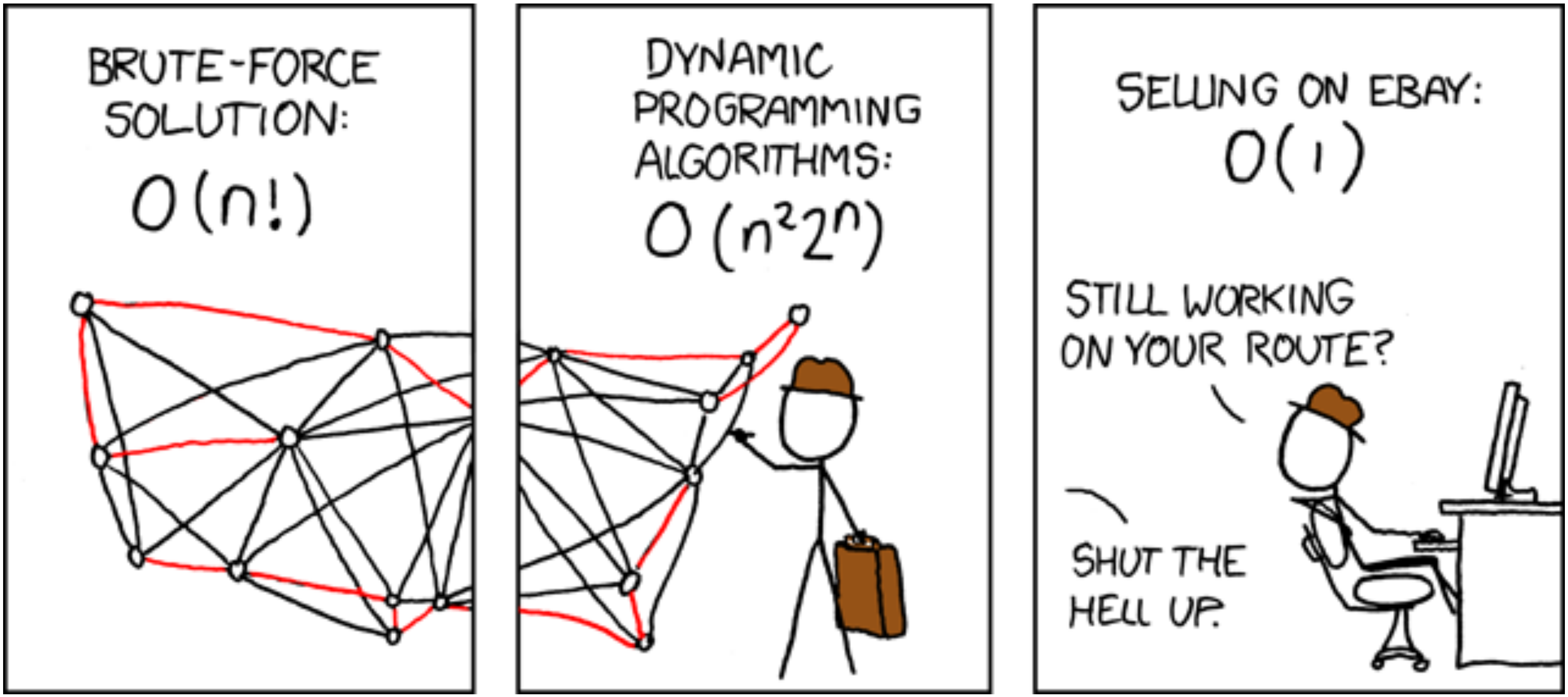
Formulating problems as search

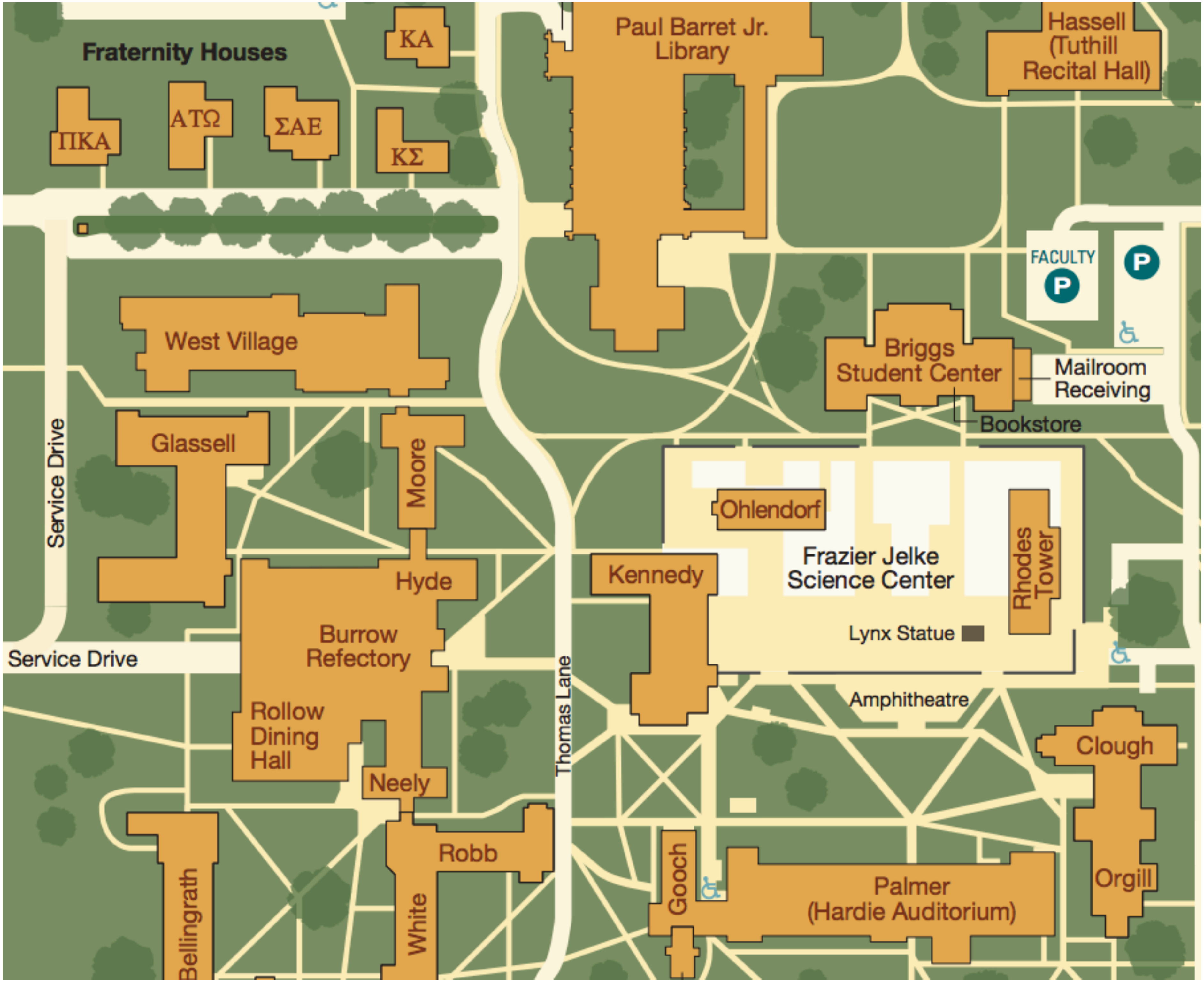
- Define:
  - What do my states look like?
  - What is my initial state? What are my goal state(s)? What does the state space "look like?" Is is a graph or a tree?
  - What is my cost function?
    - How do I know how "good" a state or action is?
    - Usually desirous to minimize, rather than maximize.
    - Usually phrased as a function of the path from the initial state to a goal state.

Formulating problems as search

- Solution:
  - A ***path*** between the initial state and a goal state.
  - ***Quality*** is measured by path cost.
  - ***Optimal solutions*** have the lowest cost of any possible path.

- State space search gives us graph searching algorithms.
- Are we searching a ***tree*** or a (true) ***graph***?





Fraternity Houses
ΚΑ
ΠΚΑ
ΑΤΩ
ΣΑΕ
ΚΣ
Paul Barret Jr. Library
Hassell (Tuthill Recital Hall)
FACULTY P
P
West Village
Briggs Student Center
Mailroom Receiving
Bookstore
Glassell
Moore
Ohlendorf
Service Drive
Frazier Jelke Science Center
Rhodes Tower
Kennedy
Hyde
Burrow Refectory
Lynx Statue
Service Drive
Rollow Dining Hall
Amphitheatre
Clough
Thomas Lane
Neely
Robb
Gooch
Palmer (Hardie Auditorium)
Orgill
Bellingrath
White

- There are two simultaneous graph-ish structures used in search:
  - (1) Tree or graph of underlying state space.
  - (2) Tree maintaining the record of the current search in progress (the ***search tree***).
- (1) does not depend on the current search being run.
- (1) is sometimes not even stored in memory (too big!)
- (2) always depends on the current search, and is always stored in memory.

Search tree

- A node n of the search tree stores:
  - a state (of the state space)
  - a parent pointer to a node (usually)
  - the action that got you from the parent to this node (sometimes)
  - the path cost g(n): cost of the path *so far* from the initial state to n.

Recap

- What things do we need to define in order to formulate a problem as a search problem?

Trees vs graphs

- If your search space is a tree, that implies there is only one path from the start state to any goal state.
  - Equivalently: only one sequence of actions for each possible goal state.

Generic search algorithms

- All search algorithms work in essentially the same manner:
- Start with initial state.
- Generate all possible successor states = expanding a node.
- Pick a new node to expand.
- Continue until we find a goal state.

Search tree

- ***Frontier:*** a data structure storing the collection of nodes that are available to be examined next in the algorithm.
  - Often represented as a stack, queue, or priority queue.
- ***Explored set:*** stores the collection of states we have already examined (and therefore don't need to look at again).
  - Often stored using a data structure that enables quick look-up for membership tests.

How do you evaluate a search strategy?

- **Completeness** — Does it always find a solution if one exists?
- **Optimality** — Does it find the best solution?
- **Time complexity**
- **Space complexity**

Uninformed search methods

- These methods have no information about which nodes are on promising paths to a solution.
- Also called: *blind search*
- Question — What would have to be true for our agent to need uninformed search?
  - No knowledge of goal location; or
  - No knowledge of current location or direction (e.g., no GPS, inertial navigation, or compass)

function TREE-SEARCH(*problem*) **returns** a solution, or failure
    initialize the frontier using the initial state of *problem*
    loop do
        if the frontier is empty **then return** failure
        choose a leaf node and remove it from the frontier
        if the node contains a goal state **then return** the corresponding solution
        expand the chosen node, adding the resulting nodes to the frontier

Frontier = stack, queue, or priority queue.

---

function GRAPH-SEARCH(*problem*) **returns** a solution, or failure
    initialize the frontier using the initial state of *problem*
    initialize the explored set to be empty
    loop do
        if the frontier is empty **then return** failure
        choose a leaf node and remove it from the frontier
        if the node contains a goal state **then return** the corresponding solution
        add the node to the explored set
        expand the chosen node, adding the resulting nodes to the frontier
            only if not in the frontier or explored set

Explored set = hash table.

Search strategies

- Breadth-first search
  - Variant — Uniform-cost search
- Depth-first search
- Depth-limited search
- Iterative deepening depth-first search

Breadth-first search

- Choose shallowest node for expansion.
- Data structure for frontier?
  – Queue (regular)
- Forbids examining the same state twice, even if on different paths. Why?
- Complete? Optimal? Time? Space?

Depth-first search

- Choose deepest node to expand.
- Data structure for frontier?
  - Stack (or just use recursion)
- Complete? Optimal? Time? Space?

Uniform-cost search

- Choose node with lowest path cost g(n) for expansion.
- Data structure for frontier?
  - Priority queue
- Suppose we come upon the same state twice. Do we re-add to the frontier?
  - Yes, if lower path cost.
- Complete? Optimal? Time? Space?

```
function UNIFORM-COST-SEARCH(problem) returns a solution, or failure
  node ← a node with STATE = problem.INITIAL-STATE, PATH-COST = 0
  frontier ← a priority queue ordered by PATH-COST, with node as the only element
  explored ← an empty set
  loop do
      if EMPTY?(frontier) then return failure
      node ← POP(frontier)   /* chooses the lowest-cost node in frontier */
      if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
      add node.STATE to explored
      for each action in problem.ACTIONS(node.STATE) do
          child ← CHILD-NODE(problem, node, action)
          if child.STATE is not in explored or frontier then
              frontier ← INSERT(child, frontier)
          else if child.STATE is in frontier with higher PATH-COST then
              replace that frontier node with child
```

Best-first search (class of algorithms)

- Same algorithm as uniform-cost search.
- Uses a different evaluation function to sort the priority queue.
- Need a heuristic function, h(n).
  - h(n) = Estimate of lowest-cost path from node n to a goal state.

A* Algorithm

- Sort priority queue by a function f(n), which should be the *estimated* lowest-cost path through node n.
- What is f?
  - $f(n) = g(n) + h(n)$

Heuristics

- A heuristic function h(n) is ***admissible*** if it never over-estimates the true lowest cost to a goal state from node n.
- Equivalent: h(n) must always be less than or equal to the true cost from node n to a goal.
- What happens if we just set h(n) = 0 for all n?

Heuristics

- A heuristic function h(n) is ***consistent*** if values of h(n) along any path in the search tree are non-decreasing.
- Equivalent: given a node n, and an action which takes you from n to node n':
  - h(n) <= cost(n, a, n') + h(n')
  - h(n) – h(n') <= cost(n, a, n')
- Consistency implies admissibility (but not the other way around).
- Difficult to invent heuristics that are admissible but not consistent.

A* Algorithm

- A* is optimal if h(n) is consistent (and therefore admissible).
  - Tree search version of A* only needs an admissible heuristic, but A* is usually used for searching graphs.

Greedy best-first search

- Use just h(n) to sort priority queue.
- Complete?
- Optimal?