L13 – Informed Search and Reward-based Formulation

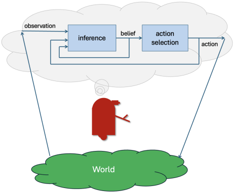
AIMA4e: Required: 3.1–3.4; 3.5.1–4; 3.6.1–2; 5.4

What you should know after this lecture

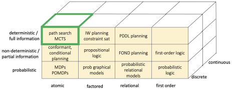
- State-space search
    - Minimizing additive path cost
    - Importance of avoiding redundant paths
- Informed search methods: GBFS and A*
- Heuristics and where to find them
- Reward-formulation problems; relation to min-cost-path
- Intro to Monte-Carlo Tree Search

Decision making!

- Given a current belief about the world
- And some objective
- What action should the agent take next?
- Apply the principle of rationality: select actions that will maximize your expected future utility



First problem setting: fully observable, deterministic



Atomic, discrete

- Agent knows:
  - State set: $\mathcal{S}$
  - Initial state: s_0
  - Action set: $\mathcal{A}$
  - Transition model: $T : \mathcal{S} \times \mathcal{A} \rightarrow \mathcal{S}$
  - Goal set: $G \subset \mathcal{S}$
  - Cost function $C : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \rightarrow \mathbb{R}$
- We need to find next action to take
- Find plan $a_1, \ldots, a_m$ from s_0 to some state in G such that $T(s_0, a_1) = s_1, \ldots, T(s_{m-1}, a_m) = s_m$ and $s_m \in G$
- Usually we try to minimize

$$\sum_i C(s_i, a_i, s_{i+1})$$

If path costs are not additive, then many algorithmic tricks don't apply and problem is much harder.

Measuring problem-solving performance

- Completeness: If there is a solution to your problem, is the algorithm guaranteed to find it?
- Cost optimality: If there is a solution, is the algorithm guaranteed to find the solution with the lowest cost?
- Computational complexity: As the size of the problem grows, how do the computation and time and space requirements grow? The answer to this depends on how we encode the input!
  - In CS algorithms tradition, problems are described as *graph search* problems, and complexity is characterized in terms of the number of vertices (states) and edges in the graph; <u>usually nearly linear in the size of the input.</u>
  - In our applications, we will often have a huge or even infinite S but it is not input to the algorithm. Instead, we provide s_0 and T, and incrementally expose the graph as we search. Characterize complexity in terms of branching factor $|A|$ and depth (also called "horizon" or "plan-length.") <u>Usually exponential in the horizon.</u>

Best-first search framework

- Critical to make a distinction between state (element of $\mathcal{S}$) and node of the search tree, which represents a path from s_0 to some state s. (Every search node has an associated state. It is possible to have multiple nodes with the same state (representing different paths to reach that state.)
- This framework takes a priority function f. Different values of f will yield different search algorithms.

Best-first search framework

```
BEST-FIRST-SEARCH(𝒮, 𝒜, s0, T, G, C, f)
n = NODE(s0)
frontier = PRIORITYQUEUE(f)
frontier.ADD(n)
reached = {s0 : n}
while not frontier.EMPTY():
    n = frontier.POP()                                   // Get node with lowest f value
    s = n.s
    if s ∈ G: return n
    for a ∈ 𝒜:                                            // Expand s
        s′ = T(s, a)
        path_cost = n.path_cost + C(s, a, s′)
        if not s′ ∈ reached or path_cost < reached[s′].path_cost:
            n′ = NODE(s′, n, a, path_cost)
            reached[s′] = n′                              // visit s′
            frontier.ADD(n′)
```

Redundant Paths

- Stupidest possible algorithm (SPA): enumerate all legal paths and pick the first one that reaches a goal state.
- There can be exponentially more paths than states!
- The *reached* data structure (sometimes called a visited list) and test in line 12 ensures that we never consider a path to a state that is higher in cost than the best one we've already found.
- In most of the searches we'll consider, in fact, we can prove that the first time we pop some path to a state off the frontier, we will have done so via a least-cost path, and so we never expand (consider the successor of) a state more than once.

Breadth-First Search

- Best-first with $f(n)$ **= number of steps in path** n
- First path found to s has fewest steps
- Can actually move the goal test earlier (from s in line 8 to s' just after line 10)
- Complete and optimal (in number of steps)
- Worst case time (and space) complexity $O(|\mathcal{A}|^d)$ where d is the length of the shortest path to the goal. This happens when there are no redundant paths.
- Note independence of $|\mathcal{S}|$
- But, complexity is also bounded by $O(|\mathcal{S}||\mathcal{A}|)$ which in some problems is smaller than $|\mathcal{A}|^d$.

Uniform Cost Search

- Best-first search with $f(n) = n.\mathit{path\_cost}$
- Assume costs are all positive.
- Like breadth-first, but pushes out frontier in equal-path-cost contours
- First path to s expanded has least cost.
- First path to s visited **does not necessarily have** least cost. See text for example of this, and why we cannot move the goal test earlier.
- Worst case time (and space) complexity

  $$O(|\mathcal{A}|^{1+\lfloor C^*/\epsilon \rfloor})$$

  where C^* is the cost of the least-cost path and ϵ is the cost of the least-cost action.
- Sometimes called Dijkstra's Algorithm (although Dijkstra's is often used to compute shortest paths to all vertices in a given finite graph.)
- As with BFS, complexity is also bounded by $O(|\mathcal{S}||\mathcal{A}| \log |\mathcal{S}|)$.

Informed state-space search methods

- Without any hints at all about how to make progress toward a goal state, we can't do better than uniform-cost search.
- A heuristic function $h : \mathcal{S} \to \mathbb{R}$ provides an estimate of the cost of the least-cost path from a state s to a goal state. (In AIMA, defined on nodes n, but really just applies to $n.s$).
- Standard example: Euclidean distance from s to a target destination in a route-finding problem.

Greedy best-first search (GBFS)

- BEST-FIRST-SEARCH where

$$f(n) = h(n.s)$$

- Always take the path out of *frontier* that we estimate has gotten closest to the goal.
- Not guaranteed to find the least-cost path!
- Often finds a satisficing (goal-reaching) path much more quickly than uniform-cost search.

A*

- Best-First-Search where

$$f(n) = n.\mathit{path\_cost} + h(n.s)$$

- Always take the path out of *frontier* that we estimate has the cheapest sum of the length of the path so far and our estimate of how for from here to the goal.
- Guaranteed to find a least-cost path if h is admissible.
- Heuristic h is admissible iff

$$h(s) \leqslant h^*(s) \quad \text{for all } s \in \mathcal{S},$$

  where $h^*(s)$ is the actual least path cost from s to a goal state.
- Heuristic h is consistent iff

$$h(s) \leqslant c(s, a, s') + h(s')$$

More about A*

- Search contours are "stretched" in the direction of goal states.
- Let C^* be cost of optimal solution path:
  - A* expands all nodes reachable from s_0 on a path where every node on the path has $f(n) < C^*$
  - A* expands no nodes with $f(n) > C^*$
- If $h(s) = h^*(s)$ then A* will not expand any nodes that are not on an optimal path.
- If $h(s)$ is close to $h^*(s)$ then there will generally not be many nodes for which $f(n) \leqslant C^*$.
- If $h(s) = 0$ then h is admissible; in this case, A* degenerates into UCS.

Heuristic Functions

- A heuristic function, ideally, is:
  - Admissible and consistent
  - Close to h^*
  - Efficient to compute
- A good source of heuristics is <u>problem relaxation</u>: make your problem "easier" in two ways:
  - Solutions have lower cost in relaxed problem
  - Solutions are faster to find in relaxed problem
- Examples:
  - Relax problem of finding a path on a road-map to finding one that can go off-road.
  - Relax problem of finding a driving route that lets you keep the car fueled to one in which you ignore fuel.
- Another strategy: <u>learn</u> h (perhaps in the form of a neural network) using supervised or reinforcement-learning based on previous experience solving related problems.

Reward-maximization formulation

Some problems are easier to formulate in terms of maximizing an amount of reward that gets accumulated over a trajectory of a fixed number of steps (horizon) H.

- Problem: $(\mathcal{S}, \mathcal{A}, T, R, H, s_0)$
- Reward instead of cost: $R : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$
- We want to find a length H path that maximizes

$$\sum_{t=0}^{H-1} R(s_t, a_t, s_{t+1})$$

- We can relax this fixed-horizon assumption later in the course, with a probabilistic model of termination.

Reduction from reward maximization to min-cost-path problem

Given reward maximization problem $(\mathcal{S}, \mathcal{A}, T, R, H, s_0)$ we can generate min-cost-path problem $(\mathcal{S}', \mathcal{A}', T', G, C, s'_o)$ so that solution to the min-cost-path problem is a solution to the original reward-maximization problem.

- $\mathcal{S}' = \mathcal{S} \times \{0, \ldots, H\}$
- $\mathcal{A}' = \mathcal{A}$
- $s'_0 = (s_0, H)$ second component is "steps to go"
- $T'((s, t), a) = (T(s, a), t - 1)$
- $G = \{(s, t) \mid t = 0\}$
- $C(s, a) = R_{max} - R(s, a)$ where $R_{max} = \max_{s,a} R(s, a)$

Note that costs are always non-negative.
We can solve using uniform-cost search!
Very hard to come up with a heuristic, since in principle, it might be possible for all the rest of your actions to pay off with R_{max} which would have a C of 0, meaning to be admissible, we need $h = 0$.

Reduction from min-cost-path to reward maximization

Given a min-cost-path problem $(\mathcal{S}, \mathcal{A}, T, G, C, s_o)$ we can generate a reward maximization problem $(\mathcal{S}', \mathcal{A}', T', R, H, s_0')$ so that solution to the min-cost-path problem is a solution to the original reward-maximization problem.

- $\mathcal{S}' = \mathcal{S} \times \{over\}$
- $\mathcal{A}' = \mathcal{A}$
- $s_0' = s_0$
- $$T'(s, a) = \begin{cases} T(s, a) & \text{if } s \notin G \text{ and } s \neq over \\ over & \text{otherwise} \end{cases}$$
- $R(s, a, s') = -C(s, a, s')$ if $s' \neq over$ else 0

Setting H is tricky:

- Could keep trying to re-solve with increasing H.
- You can do MCTS (or some other solution methods) on indefinite horizon problems, where instead of having a fixed horizon H, there are states marked as terminal and the "rollout" ends when one is reached (but you *still* need a max horizon in practice).

Monte-Carlo Tree Search

Another strategy for search guidance is to "learn" from your current search.

- Rather than systematically growing the tree, consider whole paths from s_0 to horizon
- Assumes a type of smoothness: paths with the same first action(s) will tend to have similar values
- If your problem is smooth, and, so far, paths starting with a_1 have had higher total reward than paths starting with a_2, then spend more time investigating paths starting with a_1!
- Particularly useful when no other heuristic is available and/or action space (hence branching factor) is very large.
- Used in games and probabilistic problems, as well.
- Assumes rewards in range $[0, 1]$. (Optimal policy is unchanged if we scale current rewards linearly to be in this range.)

Upper confidence bounds

Consider a situation in which you are trying to select among K actions, $a_1, \ldots, a_k$. Assume:

- You have, so far, executed N total actions
- You have, so far, executed action k for N_k trials
- The total utility you got for executing action k is U_k

What is an optimistic but realistic upper bound on the value of executing action k?

$$\text{UCB}(N, N_k, U_k) = \begin{cases} \frac{U_k}{N_k} + C\sqrt{\frac{\log N}{N_k}} & \text{if } N_k > 0 \\ \infty & \text{otherwise} \end{cases}$$

If individual utility values are in range $[0, 1]$ then a reasonable choice is $C = 1.4$. (Lots of interesting theory behind this!)

Simple MCTS example

- We first pick a_1 and get value 0.9:

$$\text{UCB}(s_0, a_1) = .9 + \sqrt{\log 1/1} \approx 0.9 \quad \text{UCB}(s_0, a_2) = \infty$$

- Pick a_2 and get value 0.1:

$$\text{UCB}(s_0, a_1) = .9 + \sqrt{\log 2/1} \approx 1.73 \quad \text{UCB}(s_0, a_2) = .1 + \sqrt{\log 2/1} \approx .93$$

- Pick a_1 and get value 0.9 again:

$$\text{UCB}(s_0, a_1) = .9 + \sqrt{\log 3/2} \approx 1.64 \quad \text{UCB}(s_0, a_2) = .1 + \sqrt{\log 3/1} \approx 1.15$$

- Pick a_1 and get value 0.9 again:

$$\text{UCB}(s_0, a_1) = .9 + \sqrt{\log 4/3} \approx 1.58 \quad \text{UCB}(s_0, a_2) = .1 + \sqrt{\log 4/1} \approx 1.28$$

- Pick a_1 and get value 0.9 again:

$$\text{UCB}(s_0, a_1) = .9 + \sqrt{\log 5/4} \approx 1.53 \quad \text{UCB}(s_0, a_2) = .1 + \sqrt{\log 5/1} \approx 1.37$$

- Pick a_1 and get value 0.9 again:

$$\text{UCB}(s_0, a_1) = .9 + \sqrt{\log 6/5} \approx 1.50 \quad \text{UCB}(s_0, a_2) = .1 + \sqrt{\log 6/1} \approx 1.44$$

- Pick a_1 and get value 0.9 again:

$$\text{UCB}(s_0, a_1) = .9 + \sqrt{\log 7/6} \approx 1.47 \quad \text{UCB}(s_0, a_2) = .1 + \sqrt{\log 7/1} \approx 1.49$$

- Woo hoo! Pick a_2! Maybe it's awesome!

Monte-Carlo Tree Search

```
MCTS(s_0, (A, T, R, H), iters)
root = NODE(s_0, horizon = H, parent = None, children = { }, U = 0, N = 0)
for iter ∈ {1, ..., iters}:
    leaf = SELECT(root)
    child = EXPAND(leaf, A, T)
    value = SIMULATE(child, A, T, R)
    BACKUP(child, value)
max_child = max(root.children, key = λn. n.U/n.N)
return root.children[max_child]        // Returns the associated action
```

```
SELECT(n)
// Follow optimistically best path through tree
if n.children
    return SELECT(max(n.children, key = λc.UCB(n.N, c.N, c.U)))
else
    return n
```

Monte-Carlo Tree Search (Cont)

```
EXPAND(n, A, T)
    // Unless remaining horizon is 0, add child nodes and return one
    if n.horizon = 0:
        return n
    else
        for a ∈ A:
            s′ = T(n.s, a)
            n′ = NODE(s′, n.horizon − 1, parent = n, children = { }, U = 0, N = 0)
            n.children[n′] = a
        return RANDOM_CHOICE(n.children)
```

```
SIMULATE(n, A, T, R)
    // Randomly finish path and return cumulative reward
    s = n.s; total_reward = 0
    for h ∈ (n.horizon, ..., 1):
        a = RANDOM_CHOICE(A)
        s′ = T(s, a)
        total_reward += R(s, a, s′)
        s = s′
    return total_reward
```

Monte-Carlo Tree Search (Cont)

```
BACKUP(n, v_below)
    // Add value v to n's statistics and pass it up
    n.N += 1
    if n.parent:
        a = n.parent.children[n]                    // Action that led to n
        v = v_below + R(n.parent.s, a, n.s)         // Value of executing a in parent
        n.U += v
        BACKUP(n.parent, v)
```

MCTS properties

- Guaranteed to (eventually) find optimal strategy with probability 1, for appropriate choice of C
- Instead of random "rollouts", you can use a semi-smart strategy, or a (learned) heuristic value function
- This is (roughly) what Alpha-Go does
- Can have poor short-term performance in cases where value function is not smooth (or short-term experience is misleading). See From Bandits to Monte-Carlo Tree Search: The Optimistic Principle Applied to Optimization and Planning, R'emi Munos, Foundations and Trends in Machine Learning, 2014.