Artificial Intelligence

Daniel Balle 2018

Classical Search

Path search problems are well-defined problems in which the solution is a *path* or action sequence from an initial state to a goal state. All possible sequences form a search tree.

All path search algorithms share the same structure, but vary primarily according to how they choose which state to expand next - their *search strategy*.

Graph Search

Graph search is a general search algorithm similar to tree search that remembers his past exploration.

```
explored = Set(); frontier = {start state}
while (frontier):
    x = frontier.pop(strategy)
    if (x is goal state): return solution
    explored.add(x)
    for (n in x.successors() if n not in explored):
        frontier.add(n)  # or update
return False
```

Graph Search Implementation

The *graph search* algorithm above can be extended to allow any search strategy by using a priority queue as the frontier. Each element in the priority should be a sequence of

- the node n itself
- the path-cost $g(n)$
- the path that explored this node

Remark. Our queue retrieves elements with lowest weights first.

Breadth-First Search

search strategy: time

BFS is a *graph search* algorithm using a FIFO queue for the frontier, thus the shallowest node is chosen for expansion at every step. The goal state test can also be applied during discovery rather than expansion.

Remark. BFS is only optimal when the path cost is a nondecreasing function of the depth of the node. It is complete under finite branching factor b.

Uniform-Cost Search

search strategy: path cost $g(n)$

UCS is a generic *graph search* algorithm which expands the node n with lowest path cost $g(n)$ first.

Remark. It is optimal for any path cost.

Remark. See *Dijkstra's* algorithm from **Algorithms** ↑ notes.

Depth-First Search

search strategy: -time

DFS is a *graph search* algorithm using a stack, it thus always expands the deepest node in the frontier.

Remark. DFS is not optimal. Its tree search isn't even complete.

Iterative Deepening DFS

Iterative Deepening search performs a *depth-limited* *depth-first search* by gradually increasing the limit.

Remark. Optimality and completeness as BFS.

Best-First Search

search strategy: $f(n)$

Best-First Search is a general *informed graph search* algorithm using an evaluation function $f(n)$ as its search strategy, usually involving a *heuristic* function:

$$h(n) = \text{estimated cost of optimal path from } n \text{ to goal}$$

Greedy Best-First Search

search strategy: $f(n) = h(n)$

Greedy Search is an *informed search* algorithm that expands nodes that are estimated to be the closest to the goal using some heuristic function $h(n)$.

Remark. It is neither optimal nor complete.

A* Search

search strategy: $f(n) = g(n) + h(n)$

A* Search is an *informed search* algorithm using an estimated cost of the cheapest solution through n as its evaluation function.

Remark. It is both complete and optimal when h is *admissible* for tree search, or *consistent* for graph search. A* is also optimally efficient in terms of nodes expanded.

Iterative Deepening A*

IDA* is an application of *iterative deepening* to *A* search* using a cutoff value for the evaluation function $f(n)$. At each iterations the new cutoff value is the lowest $f(n)$ that previously exceeded the cutoff.

Heuristic Admissibility

A heuristic h is admissible if it never overestimates the true optimal path from n to the goal $h^*(n)$.

$$h(n) \leq h^*(n) \quad \forall n \in V$$

Heuristic Consistency

A heuristic h is consistent if for any successors n' of n reached through action a, it holds that:

$$h(n) \leq c(n, a, n') + h(n') \quad \forall (n, n') \in E$$

Remark. Consistency implies *admissibility*.

Path Search Todo

- Runtime and Memory properties
- Optimality proof of A*
- Recursive Best-First Search, MA*, SMA*

Optimization and Local Search

Local search algorithms operate using a single current node and solve optimization problems whose solution is a *state* $s^* \in S$ according to some objective function $f(s)$.

Hill-Climbing Search

$$x^{(t+1)} \leftarrow x^{(t)} + \eta \nabla f(x^{(t)}) \qquad \eta > 0$$

Hill-climbing search is a greedy *local search* algorithm which continually moves in the direction of increasing value. Variations include:

Stochastic: choose randomly among all uphill moves.

First-choice: randomly generate successors until a better state than the current one is found.

Random-restart: Multiple tries with random initial states.

Simulated Annealing

This *local search* algorithm picks a random move n:

- if the move improves the state, it accepts
- otherwise it accepts with probability $p = e^{\Delta E/T}$

with ΔE the amount by which the state is worsened, and T the temperature at time t.

Local Search Todo

- Local Beam Search
- Genetic Algorithms
- Continuous local search
- Constraint Search Problem, Integer Programming
- AND-OR Search
- Online vs Offline Search
- All the rest ...

Adversarial Search

Games are adversarial search problems in a competitive multi-agent environment whose solution is a *strategy* $f : S \to A$.

Similar to search, the sequences of actions from the initial game state generate a *game tree*. A utility functions defines the final value of a terminal state for each player.

Remark. Games can also be seen as *decision problems*.

Minimax Decision

The minimax decision in a given game state s is choosing the optimal action a, assuming opponents play optimally. We maximize the worst-case outcome, based on the *minimax value*:

$$mv(s) = \begin{cases} \text{utility}(s) & \text{if } s \text{ is terminal} \\ \max_a \, mv(s+a) & \text{if player is } Max \\ \min_a \, mv(s+a) & \text{if player is } Min \end{cases}$$

Minimax Algorithm

Given a current state s, the minimax algorithm performs the *minimax decision*. This results in a complete *depth-first* search exploration of the game tree.

```
decisions = {
    action : minimaxValue(game.getSuccessor(state, action))
    for action in game.getActions(state)
}
return max(decisions, key=decisions.get)
```

Remark. In practice we use a *ply* limited search: *heuristic minimax*.

Alpha Beta Pruning

α-β pruning is an optimization technique for *minimax search* which prunes away branches in the minimax tree that won't change the final decision, using additional variables:

- α = value of best decision by Max on path to root
- β = value of best decision by Min on path to root

Alpha Beta Implementation

The *α-β pruning* algorithm `minimaxValue(s, α, β)` for player *Min* can be implemented as follows.

```
for next in game.getSuccessors(state):
    value = min(value, minimaxValue(next, α, β))
    if (value < α) : return value
    β = min(β, value)
return value
```

Because *Max* can choose an action yielding α, this subtree won't be considered for his final decision if we yield a value $< \alpha$.

Heuristic Minimax

H-minimax is a depth or ply limited *minimax search* using a heuristic evaluation function for non-terminal states.

$$hm(s,d) = \begin{cases} \text{eval}(s) & \text{if cutoff}(s,d) \\ \max_a \ hm(s+a, d+1) & \text{if player is } \mathit{Max} \\ \min_a \ hm(s+a, d+1) & \text{if player is } \mathit{Min} \end{cases}$$

Remark. This is used in practice for imperfect real-time decisions. Another approach is using *iterative deepening* until time runs out.

Expecti-Minimax Value

Expecti-minimax is an extension of *minimax* to stochastic games in which we use the expected value for states played by a *chance* player.

$$em(s) = \begin{cases} \sum_a p(a) em(s+a) & \text{if player is } Chance \\ mv(s) & \text{otherwise} \end{cases}$$

Bayesian Networks

A Bayesian network is a locally structured representation of a *joint probability* distribution as a directed acyclic graph where:

- each node X is a random variable
- annotated with *conditional* probability distribution given its incoming nodes or parents $\mathbf{P}(X|\text{Parents}(X))$

Each node X is conditionally independent of its *non-descendants* given his parents, and all other nodes given his *Markov Blanket*.

Markov Blanket

In a *Bayesian network*, the Markov blanket ∂X of a node X is the set including his *parents*, *children*, and *children's parents*.

$$X \perp\!\!\!\perp B \mid \partial X \quad \text{or} \quad P(X \mid \partial X, B) = P(X \mid \partial X)$$

Chain Rule

The chain rule can be used to compute any *joint* distribution given only conditional probabilities, e.g. by a *Bayesian network*:

$$P(X_1, ..., X_n) = \prod_{i=1}^{n} P(X_i \mid X_{i-1}, ..., X_1)$$

D-Separation

In a *Bayesian network*, a path p *d-separates* X and Y by set Z containing variable M if *any* of these conditions hold:

- p contains a directed chain $X \rightarrow ... \rightarrow M \rightarrow ... \rightarrow Y$
- p contains a fork $X \rightarrow ... \rightarrow M \leftarrow ... \leftarrow Y$
- p contains an inverted fork $X \leftarrow ... \leftarrow N \rightarrow ... \rightarrow Y$, where neither N nor any of its descendants $\in Z$

If *all* paths p from X to Y are d-separated, then $X \perp\!\!\!\perp Y \mid Z$.

Probabilistic Inference

Probabilistic inference computes the *posterior* distribution of a set of *query* variables given an assignment of *evidence* variables.

$$\mathbf{P}(X \mid \mathbf{e})$$

Remark. We usually denote by $\mathbf{Y}$ the *hidden* variables.

Inference by Enumeration

IE is an exact *inference* technique which selects the terms consistent with the evidence $\mathbf{e}$ from the *joint* distribution to sum out the hidden variables before normalizing.

$$\mathbf{P}(X \mid \mathbf{e}) = \alpha \mathbf{P}(X, \mathbf{e}) = \alpha \sum_{\mathbf{y}} \mathbf{P}(X, \mathbf{e}, \mathbf{y})$$

Remark. Given a *Bayesian network*, $\mathbf{P}(X, \mathbf{e}, \mathbf{y})$ can be computed using the *chain rule* on the conditional probabilities.

Variable Elimination

VE is a dynamic programming algorithm for exact *inference* summing out the hidden variables $\mathbf{y}$ right-to-left and storing intermediate *factors* $\mathbf{f}_i$.

$$\mathbf{P}(X \mid \mathbf{e}) = \alpha \sum_{y_1} \mathbf{f}_2(y_1, X, \mathbf{e}_2) \times \sum_{y_2} \mathbf{f}_1(y_2, X, \mathbf{e}_1)$$

Remark. $\times$ is point-wise product operator as $\mathbf{f}_i$ are matrices.

Direct Sampling

Direct or prior sampling for *Bayesian networks* is a randomized sampling algorithm which samples each variable in *topological* order, conditioned on the sampled values of its parents.

```
for X_i in Topologic[X_1, ..., X_n] do
    x[i] ← random sample from P(X_i|parents(X_i))
```

The *consistent* estimate of the *joint* probability is then given by

$$S_{PS}(\mathbf{x}) = N_{PS}(\mathbf{x}) \cdot N^{-1} \xrightarrow{N \to \infty} P(\mathbf{x})$$

Remark. This is an instance of a *Monte Carlo* algorithm.

Rejection Sampling

Rejection Sampling performs *prior sampling* and then rejects all samples that do not match the evidence $\mathbf{e}$.

$$\hat{\mathbf{P}}(X|\mathbf{e}) = \mathbf{N}_{RS}(X, \mathbf{e}) \cdot N_{PS}(\mathbf{e})^{-1}$$

Remark. Rejection sampling produces a consistent estimate.

Likelihood Weighting

Likelihood weighting is an extension to *prior sampling* which samples only the non-evidence variables, and weighs each sample according to its *likelihood* given the evidence.

```
for X_i in Topologic[X_1, ..., X_n] do
    if X_i is evidence in e with value x_i do
        x[i] ← x_i ; w ← w · P(X_i = x_i | parents(X_i)) # initially 1
    else x[i] ← random sample from P(X_i | parents(X_i))
```

$$\hat{P}(x|\mathbf{e}) = \alpha \sum_{\mathbf{y}} N(x, \mathbf{y}, \mathbf{e}) \cdot w(x, \mathbf{y}, \mathbf{e})$$

Remark. This is an instance of importance sampling.

Gibbs Sampling

From an arbitrary sample s_1 with correct evidence $\mathbf{e}$, generate the next sample s_2 by sampling one of the non-evidence variables X_i conditioned on the values of its *Markov Blanket* ∂X_i in s_1.

$$x_i \sim \mathbf{P}(X_i \mid \partial X_i \in s_1)$$

Remark. This is a *Markov Chain Monte Carlo* algorithm.

Stochastic Decision Problems

A solution to a sequential decision problem in a stochastic environment is a *policy* mapping every state s to an action a:

$$\pi : S \to A$$

Remark. Partially observable environments require a mapping from belief states b instead $\pi : \Delta(S) \to A$.

Policy and Utility

We denote by $V^*(s)$ or $U^*(s)$ the expected sum of future rewards, or *utility*, when acting optimally starting from state s. The utility of acting under policy π is denoted as $V^\pi(s)$. An *optimal policy* π^* is one that yields the highest expected utility:

$$\pi_s^* = \arg\max_\pi V^\pi(s)$$

Planning vs. Learning

Algorithms to solve *decision problems* can be categorized based on the environment:

Fully specified $\rightarrow$ we use *offline* algorithms to *plan* π

Unknown $\rightarrow$ we use *online* algorithms to *learn* π

Remark. Learning is sometimes referred to as *online* planning.

Markov Decision Process

An MDP is a mathematical framework for non-deterministic decision *planning* problems. It consists of:

- a set of states S, a start state s_0 and terminal states $\subseteq S$
- a set of actions $A(s)$ for each state $s \in S$
- a *Markovian* (memoryless) transition model $P(s'|s, a)$
- a reward function $R(s, a, s')$ and a discount factor γ

Finite Horizons and Discounting

To prevent policies π yielding infinite utility we introduced finite horizons or discounting. The ladder results in *discounted rewards* $\gamma^t R(s, a, s')$ for actions taken at time t. We then obtain a discounted utility function:

$$U(\mathbf{h}) = R(s_0) + \gamma R(s_1) + \gamma^2 R(s_2) + ... < \frac{R_{max}}{1-\gamma}$$

Remark. Finite horizon decision problems yield *non-stationary* policies, i.e. policies depending on time $\pi : S \times T \to A$.

Bellman Equation and Optimality

An optimal policy π^* is one that satisfies the *Bellman Equation*:

- $V^*(s) = \max_a \sum_{s' \in S} P(s'|s,a) \{R(s,a,s') + \gamma V^*(s')\}$
- $Q^*(s,a) = \sum_{s' \in S} P(s'|s,a) \{R(s,a,s') + \gamma V^*(s')\}$
- $\pi^*(s) = \arg\max_a Q^*(s,a)$

Remark. The final line is called *Policy Extraction.*

Value Iteration

Value Iteration is an DP algorithm for computing the values $V^*(s)$ using the Bellman update until convergence:

$$V^{k+1}(s) = \max_a \sum_{s' \in S} P(s'|s,a) \left\{ R(s,a,s') + \gamma V^k(s') \right\}$$

Remark. Equivalent to a depth-k expectimax search on the Markov Decision Process search tree.

Policy Iteration

Policy Iteration is an algorithm for computing the optimal policy π^* directly using an initial policy π_0 and alternating:

1. π **evaluation**: compute $V^{\pi_i}(s)$ for each state.
2. π **improvement**: $\pi_{i+1}(s) = \arg\max_a Q^{\pi_i}(s, a)$

Remark. π evaluation can be done using a simplified value iteration algorithm or solving $|S|$ linear equations using algebra.

Simplified Value Iteration

This algorithm uses a simplified Bellman update to evaluate a given policy π for Policy Iteration.

$$V_{k+1}^{\pi}(s) = \sum_{s' \in S} P(s'|s, \pi(s)) \left\{ R(s, \pi(s), s') + \gamma V_k^{\pi}(s') \right\}$$

Game Theory

Uncertainty in an environment can be caused by non-deterministic actions or by the presence of multiple agents in the environment. In the contest of games we talk about *strategies* rather than policies.

See the **Game Theory** ↑ notes for more details on decision problems in multi-agent environments.

Reinforcement Learning

RL is an *online* planning process for unknown stochastic decision problems which *explores* to collect *reinforcements*.

Passive $\rightarrow$ learn the state values $V^{\pi_0}(s)$ of a fixed policy π_0

Active $\rightarrow$ learn and *use* an estimate of the optimal policy π^*

Machine and Statistical Learning

See the **Computational Statistics** [↑] and **Data Mining** [↑] notes for more details on *supervised* and *unsupervised* learning.

Exploration vs Exploitation

Active RL algorithms use an *exploration* policy π_e which captures a trade-off between:

Exploration $\rightarrow$ improve the estimate of the true model

Exploitation $\rightarrow$ maximize rewards according to $\hat{\pi}^*$

Remark. Passive learning for π_0 is equivalent to active learning with constant exploration policy $\pi_e = \pi_0$.

Direct Utility Learning

Passive & model-free RL

- Follow a fixed policy π_0 to collect pairs $(s, V_k^{\pi_0}(s))$ where $V_k^{\pi_0}(s)$ is the sum of rewards received in episode k from state s.
- Estimate $V^{\pi_0}(s)$ by averaging all $V_k^{\pi_0}(s)$ containing s.

Remark. Similar to supervised learning for $\{s \to V_k^{\pi_0}(s)\}$ data.

Adaptive Dynamic Programming

Active & model-based RL

- collect samples $(s, \pi_e(s), s', R)$ through π_e
- estimate the reward $\hat{R}$ and transition function $\hat{T}$
- use *offline* planning to compute $\hat{V}^{\pi_0}(s)$ or $\hat{\pi}^*$

Remark. Similar to supervised learning for $\{(s, a) \to (s', R)\}$ data.

Temporal Difference Learning

Passive & model-free RL

Under fixed π, update $\hat{V}^\pi(s)$ for every sample $R(s, \pi(s), s')$ using an exponential running average with *learning rate* α:

$$\Omega \triangleq R(s, \pi(s), s') + \gamma \hat{V}^\pi(s')$$
$$\hat{V}^\pi(s) \leftarrow (1-\alpha)\hat{V}^\pi(s) + \alpha\Omega = \hat{V}^\pi(s) + \alpha(\Omega - \hat{V}^\pi(s))$$

Q-Learning

Active & model-free RL

Learn the q-state values $\hat{Q}(s, \pi_e(s))$ for every sample $R(s, \pi_e(s), s')$ using a running average with *learning rate* α:

$$\Omega \triangleq R(s, a, s') + \gamma \max_{a'} \hat{Q}(s', a')$$
$$\hat{Q}(s, a) \leftarrow (1 - \alpha)\hat{Q}(s, a) + \alpha\Omega = \hat{Q}(s, a) + \alpha(\Omega - \hat{Q}(s, a))$$

Remark. max over q-state values $\implies$ this is *off-policy* learning.

Epsilon Greedy

ϵ-Greedy is an exploration policy π_e for active RL which explores with (decaying) probability ϵ and exploits otherwise.

$$\pi_e(s) = \begin{cases} \text{random}\{A(s)\} & \text{with probability } \epsilon \\ \hat{\pi}^*(s) & \text{otherwise} \end{cases}$$

Remark. A greedy agent uses 0-greedy exploration. Using $\epsilon > 0$ guarantees convergence to true utilities and optimal policy.

Exploration Function

An exploration function $f(s, a)$ encourages exploration by assigning optimistic values $Q^+(s, a)$ to q-states with low attempts $N(s, a)$.

$$Q^+(s,a) = \sum_{s' \in S} P(s'|s,a)\{R(s') + \max_{a'} f(s', a')\}$$
$$f(s,a) = Q^+(s,a) + k \cdot N(s,a)^{-1} \qquad \text{(example)}$$

Remark. Then the exploration policy is just $\pi_e = \hat{\pi}^*$. This yields faster convergence to the optimal policy, but not values.

Exploration Function for Q-Learning

An exploration function $f(s, a)$ can be integrated into the q-learning update as follows:

$$\Omega \triangleq R(s, a, s') + \gamma \max_{a'} f(s', a')$$
$$Q^+(s, a) \leftarrow (1 - \alpha)Q^+(s, a) + \alpha\Omega$$

Approximate RL

Function approximation for *model-free* RL uses feature representations $\mathbf{f}(s)$ or $\mathbf{f}(s,a)$ to *learn* a weight vector $\mathbf{w}$.

$$\hat{V}(s) = \mathbf{w}^\top \mathbf{f}(s) \qquad \hat{Q}(s,a) = \mathbf{w}^\top \mathbf{f}(s,a)$$

Remark. Approximate reinforcement learning in a fully observable environment is similar to *supervised learning*.

Widrow-Huff Rule

The Widrow-Huff rule for *online least squares* can be used learn weights $\mathbf{w}$ of $\hat{G}(x) = \mathbf{w}^\top \mathbf{f}(x)$ with every sample $g(x)$ for some x

$$E(x) = (\hat{G}(x) - g(x))^2/2$$

$$w_i \leftarrow w_i - \alpha \cdot \partial E(s)/\partial w_i = w_i + \alpha(g(x) - \hat{G}(x))f_i(x)$$

Approximate TD Learning

Using feature representation $\mathbf{f}(s)$ for states, update weight vector $\mathbf{w}$ for every sample $R(s, a, s')$:

$$\Delta \triangleq R(s, a, s') + \gamma \hat{V}(s') - \hat{V}(s)$$
$$w_i \leftarrow w_i + \alpha \Delta f_i(s)$$

Remark. Widrow-Huff with $g(s) = R(s, a, s') + \gamma \hat{V}(s')$.

Approximate Q-Learning

Using feature representation $\mathbf{f}(s, a)$ for q-states, update weight vector $\mathbf{w}$ for every sample $R(s, \pi_e(s), s')$:

$$\Delta \triangleq R(s, a, s') + \gamma \max_{a'} \hat{Q}(s', a') - \hat{Q}(s, a)$$
$$w_i \leftarrow w_i + \alpha \Delta f_i(s, a)$$

Remark. Widrow-Huff with $g(s, a) = R(s, a, s') + \max_{a'} \hat{Q}(s', a')$.

Todo

- Genetic Algorithm
- Neural Networks
- Multi-Layer Perceptron
- Back-propagation
- SVM
- Gradient Boosting Machines
- CNNets
- K-Means
- Logistic Regression

MNIST in Keras

Loading and inspecting the *MNIST* dataset in Python using packages `keras` and `Tensorflow` are done as follows.

```
from keras.datasets import mnist
(train_images, train_labels),
    (test_images, test_labels) = mnist.load_data()
```

```
plt.imshow(train_images[0], cmap=plt.cm.binary)
plt.show()
```

Remark. For Mac OS `10.10` use `tf == 1.4.0` and `keras == 2.1.3`.

Softmax Function

The softmax or *normalized exponential* function σ is a generalization of the *logit function* which "smashes" an arbitrary vector to a categorical distribution, $\sigma : \boldsymbol{z} \in \mathbb{R}^K \to (0,1)^K$.

$$\sigma(\boldsymbol{z})_i = \frac{e^{z_i}}{\sum_j e^{z_j}}, \qquad \sum_i \sigma(\boldsymbol{z})_i = 1$$

Remark. σ is used is *multiclass classification*, linear discriminant analysis, naive Bayes classifiers, and artificial neural networks.