CS2109S
Introduction to Artificial Intelligence and Machine Learning

AY2021/22 Semester 2

Notes by Jonathan Tay

Last updated on February 12, 2022

Contents

1 Informed Search

Informed search involves the use an **evaluation function** $f(n)$ which estimates desirability for every node in the search tree.

These search algorithms can therefore **find solutions more efficiently** by expanding the most desirable unexpanded nodes first.

A **heuristic function** $h(n)$ estimates the cost from state n to a goal state. Heuristics form a component of the evaluation function.

1.1 Greedy Best-first Search

Greedy best first search is a special case of best-first search which always **expands the node that appears closest to the goal** — the node with the lowest $h(n)$ value.

This algorithm is usually implemented with a **priority queue**, with nodes in the frontier **ordered in decreasing order of desirability**. The evaluation function $f(n) = h(n)$.

Greedy best-first search is **not complete**, as it can get stuck in loops.

It is **not optimal** either, since it tries to get as close to a goal as possible at each iteration, but this greediness can lead to a solution with a higher cost. *In general, greedy algorithms are not optimal.*

It has a **time complexity of $O(b^m)$**, but a good heuristic function can reduce this significantly to $O(bm)$.

Likewise, it has a **space complexity of $O(b^m)$** since it needs to store every node in memory.

1.2 A* Search

A* search builds on best-first search by **avoiding the expansion of expensive nodes**, using an evaluation function $f(n) = g(n) + h(n)$.

$f(n)$ estimates the total cost of the path from n to a goal. $g(n)$ is the total cost to reach n from the initial state, while $h(n)$ **estimates** the cost from n to a goal.

A* search is **complete** as long as the state space is finite.

It is **optimal** as long as its heuristic is **admissible**. *An admissible heuristic never overestimates the cost to reach a goal, and is therefore optimistic.*

Formally, a heuristic is admissible if $\forall n(h(n) \leq h^*(n))$, where $h^*(n)$ is the true cost to reach a goal state from node n.

If $h(n)$ is admissible, `TREE-SEARCH` is optimal.

Furthermore, a heuristic is consistent if, for every node n and every successor n' generated by an action a, $h(n) \leq \text{cost}(n, a, n') + h(n')$.

If $h(n)$ is consistent, `GRAPH-SEARCH` is optimal.

Every consistent heuristic is admissible, but not vice versa. Therefore, with a consistent heuristic, A* search is optimal as well.

A* search has a **time complexity of $O(b^m)$**. However, A* search can **prune** away nodes not required for finding an optimal solution, *such that it is much more efficient than uninformed searches despite its exponential complexity.*

It also has a **space complexity of $O(b^m)$**, since every node needs to be stored in memory.

1.3 Memory-bounded Heuristic Search

With memory being the main issue with A* search, other algorithms exist with tricks to reduce the space complexity.

1.3.1 Iterative-deepening A* Search

IDA* search is to A* what iterative-deepening search is to depth-first search.

Instead of setting the cutoff based on depth, IDA* uses the f-cost of $g(n)+h(n)$, incrementing the cutoff to the best f-cost which exceeded the cutoff on the previous iteration.

1.3.2 Recursive Best-first Search

RBFS resembles the typical best-first search, but keeps track of the best alternative f-value before exploring the current node.

If the current node exceeds this limit, it recursively backtracks to explore the alternative path. While backtracking, it replaces the f-value of nodes along the way with the best f-value of that node's children.

1.3.3 Simplified Memory-bounded A* Search

A* search uses too much memory, while both IDA* and RBFS use too little memory — IDA* keeps only the f-cost limit, while RBFS uses linear space, which under-utilizes memory even if more is available.

SMA* improves on this by expanding the best node until all memory is used up.

Then, it drops the node with the highest f-value, backtracking and replacing that node's parent's f-value with the dropped node's f-value before continuing.

This algorithm is not perfect, however. On very hard problems, SMA* can switch back and forth between alternative paths, each time running out of memory and repeatedly regenerating the same nodes.

1.4 Formulating Heuristics

A search algorithm using a more efficient heuristic will never expand more nodes than one with a less efficient heuristic.

Given two admissible heuristics, $h_1(n)$ and $h_2(n)$, h_2 **dominates** h_1, if for any node n, $h_2(n) \geq h_1(n)$.

Therefore, it is better to use a heuristic with higher $h(n)$ values, since it will expand fewer nodes as long as it is consistent.

A **relaxed problem** is a modified problem with fewer restrictions placed on its actions. Any solution for the orgiginal problem also solves the relaxed problem.

The cost of an optimal solution to a relaxed problem is an admissible and consistent heuristic for the original problem.

1.5 Using Inadmissible Heuristics

If we are willing to to accept suboptimal solutions in exchange for fewer nodes explored, **inadmissible heuristics** can be used.

Inadmissible heuristics may overestimate costs, risk missing the optimal solution, but if they are more accurate, they can lead to fewer nodes explored.

This is especially useful when the state-space is large, and optimality is not a requirement.

2 Local Search

Local search algorithms search unsystematically from an initial state to its neighbors *without keeping track of paths and previous states.*

They are useful in **optimization problems** where finding the best possible state is important, and the path to get there is irrevelant.

2.1 Hill-climbing Search

On each iteration, **hill-climbing search** keeps track of one current state, and moves to the neighboring state with the **steepest ascent**.

It does not look beyond neighboring states, terminating at a peak, which means it can get stuck on **local maxima** and **plateaus**.

A solution to plateaus is allowing **sideways movement**, while also restricting the number of these moves so that the algorithm does not wander the plateau forever.

When there are many successors to any given state, choosing the **first-better-choice** is preferable. Another variant is **stochastic hill climbing**, which chooses a better successor randomly.

2.2 Simulated Annealing

Hill-climbing algorithms get stuck on **local maxima** since they never make downhill moves, while **random walk** algorithms will eventually reach the global maxima, albeit inefficiently.

By combining the two, doing something random once in a while allows for breaking out of local maxima, *while yielding both efficiency and completeness.*

Simulated annealing algorithms pick a random move each iteration, accepting it if it improves the situation. Otherwise, it is accepted with a probability inversely and exponentially proportional to its "badness".

Also, simulated annealing algorithms become less likely to accept bad moves as time goes on. *With enough time, it will find a global maxima with probability approaching 1.*

2.3 Beam Search

Rather than just keeping one node in memory, **beam search** performs k hill-climbing searches in parallel.

In **local beam search**, information can be shared between parallel search threads, allowing the algorithm to abandon unfruitful searches.

However, this means that all k states may end up clustered together in the state space, effectively making it a k-times slower hill-climbing search. To combat this, **stochastic beam search** randomly chooses successors independently instead.

2.4 Genetic Algorithms

In **genetic algorithms**, successor states are generated from a **recombination** of two parent states at random **crossover points**, and **random mutations** are allowed.

Not all successors are kept — **selection processes** evaluate individuals with a **fitness function**, **culling** those that do not meet a fitness threshold.

2.5 Online Search

So far, the algorithms have all been for **offline search**, in which the algorithms compute a complete solution and then execute it.

Online search, however, interleaves the computation and execution. It is useful in dynamic environments where the agent is penalized for extended computation.

It is also necessary in **exploration problems**, where the agent is clueless about the states and actions of its environment, and new observations are available only after acting.

3 Adversarial Search

Adversarial search problems arise from **competitive environments** wherein two or more agents have conflicting goals, such as in games.

Even if we could specify every possible reply to an opponent's move, it is often computationally intractable to do so. For this topic, we will only consider two-player games.

3.1 Two-player Games

In two-player games, **"move"** is a synonym for "action", and **"position"** is a synonym for "state".

A **ply** is used to mean a move by one player, increasing the depth of the **search tree** by one. This disambiguation is useful in games where a move means both players have acted.

Formally, games are defined with the following elements:

- S_0, the **initial state** of the game at the start.
- `TO-MOVE(`s`)`, the player whose turn it is to move in state s.
- `ACTIONS(`s`)`, the set of legal moves in state s.
- `RESULT(`s`, `a`)`, the **transition model** from s to s' after taking action a.
- `IS-TERMINAL(`s`)`, a **terminal test** whether the game is over (at **terminal states**).
- `UTILITY(`s`, `p`)`, a **utility function** returning a numeric value for player p when the game ends in terminal state s.

3.2 Minimax Search

Minimax search is used in two-player games where one player is assigned `MAX` and the other `MIN`.

An optimal strategy can be determined by working out the **minimax value** of each state in the tree, where the players play optimally.

In every non-terminal state, `MAX` always seeks to maximize the minimax value, while `MIN` minimizes.

Minimax search is **complete** if the search tree is finite.

It is **optimal** only against an optimal player.

The **time complexity is $O(b^m)$** since it has to explore the entire game tree depth-first. Since the game is deterministic, with depth-first exploration, it has a **space complexity of $O(bm)$**.

However, the exponential time complexity makes it computationally intractable for complex games such as chess, with an average game having approximately 10^{123} states.

3.3 Alpha-Beta Pruning

We can optimize the computation a minimax value by keeping track of the minimum and maximum value seen thus far, ignoring paths which have no effect on the outcome of the game.

By **pruning** large parts of the game tree, we will not have to examine every single state, reducing the exponent in the time complexity.

3.3.1 Example of α–β Pruning

Consider a state s in which it is `MAX`'s turn to choose between two actions leading to s_A and s_B. We begin with an α–β range of $[-\infty, +\infty]$.

At s_A, when it is `MIN`'s turn, `MIN` will always choose the action with the minimum minimax value. Let this value be k.

Now, we know that the minimax value at the root node s is *at least* k, so we can update the α–β range to $[k, +\infty]$.

Then, we evaluate the minimax value for s_B. If we encounter any leaf node of S_B with a minimax value *less than* k, going down the path of s_B is a poor choice, since s_A guarantees a minimax score of *at least* k.

Therefore, the optimal path for `MAX` at node s is to go down s_A.

We did not have to consider the other leaf nodes of s_B once we had found a leaf node of s_B with value *less than* k, allowing us to **prune** the search tree!

3.3.2 Move Ordering

The order in which states are explored affects the effectiveness of α–β pruning.

With perfect ordering, only $O(b^{\frac{m}{2}})$ nodes need to be examined as compared to $O(b^m)$ by minimax.

Therefore, we can **double the search depth** at best, and at worst α–β pruning has no effect. Ultimately, **pruning does not affect the final result**.

3.4 Resource Limits

Computation time is usually limited. Eventually, we have to cut off the search when we are unable to reach all terminal states.

We therefore replace `IS-TERMINAL` with `IS-CUTOFF` with a certain depth limit, for example.

Consequently, we need a heuristic **evaluation function** `EVAL` which estimates the utility of a state, to replace the utility function `UTILITY`. Usually, it is a **weighted linear sum** of the various **features** of a state.

We can also deal with repeated states in `GRAPH-SEARCH` with **memoization**.

4 Linear Regression

Regression problems are one of the major classes of **supervised machine learning**. In regression problems, we are given data with m training examples, an input with x variables (**features**), and are tasked with predicting the output variables y.

Univariate linear regression is essentially "*fitting a straight line*".

It is defined by the **hypothesis function** $h_\theta(x) = \theta_1 x + \theta_0$. The θ_0 and θ_1 coefficients are known as **weights**.

The objective is to find the best h_θ that best fits the data well. *The line that best fits is the one with weights that minimizes the cost (mean squared error)* (L_2 loss), **without overfitting**.

By Occam's razor, we prefer the simplest hypothesis that is consistent with the data.

4.1 Cost Function

$J(\theta_0, \theta_1) = \frac{1}{2m}\sum_{i=1}^{m}(h_\theta(x_i) - y_i)^2$

The **cost function** is the mean of the squared errors between the data y_i and the hypothesis, as predicted by $h_\theta(x_i)$, over the m training examples in the data.

For simplicity, we often use the following simplified version of the squared errors function:

$J'(\theta_1) = \sum_{i=1}^{m}(\theta_1 x_i - y_i)^2$

Squared error, unlike absolute error (L_1 loss) is differentiable everywhere, and is more amenable to mathematical optimization. It also penalizes larger errors more heavily.

4.2 Gradient Descent

On a contour plot, the cost function is **convex**. We can minimize error by iteratively updating the weights θ_0 and θ_1 in the direction of the gradient until convergence — a method called **gradient descent**.

During each iteration, the weights θ_j are updated **simultaneously** as follows:

$\theta_j := \theta_j - \alpha\frac{\partial J(\theta_0, \theta_1, \dots)}{\partial \theta_j}$

$\implies \theta_0 := \theta_0 - \alpha\frac{1}{m}\sum_{i=1}^{m}\big(h_\theta(x_i) - y_i\big)$

$\implies \theta_1 := \theta_1 - \alpha\frac{1}{m}\sum_{i=1}^{m}\big(h_\theta(x_i) - y_i\big)\cdot x_i$

The parameter α is called the **learning rate**, and is always positive. Since the error is quadratic, the partial derivative will be linear, and at every step, *the cost decreases at a decreasing rate toward the minimum.*

Care must be taken when choosing the learning rate. *It is important to choose a value that is small enough that the algorithm converges quickly, but large enough that the algorithm does not overshoot the minimum.*

4.3 Variants of Gradient Descent

The gradient descent covered earlier is known as **batch** (or deterministic) **gradient descent**.

Summing all m training examples in every step (also known as an **epoch**), may be very slow when m is large.

Stochastic gradient descent is faster, randomly sampling just one of the m training examples at each step. Alternatively, **minibatch gradient descent** samples a subset of the m training examples.

4.4 Multivariable Linear Regression

Regression problems are often multivariate, having more than one feature. We can extend x to n features with an n-dimensional vector.

The **hypothesis function** then becomes:

$h_\theta(x) = \theta_0 x_0 + \theta_1 x_1 + \cdots + \theta_n x_n = \theta^T x$

$\theta^T x$ is the dot product of the weights with the features. Since the w_0 term is the intercept, we can fix x_0 to 1.

The **cost function** becomes:

$J(\theta) = \frac{1}{2m} \sum_{i=1}^{m} (h_\theta(x_i) - y_i)^2$

Gradient descent for multivariable linear regression is similar to the univariate case, except that the x_i are now n-dimensional vectors:

$\theta_j := \theta_j - \alpha \frac{\partial J(\theta)}{\partial \theta_j}$

Therefore, for n features,

$\theta_0 := \theta_0 - \alpha \frac{1}{m} \sum_{i=1}^{m} \big(h_\theta(x_i) - y_i\big) \cdot x_{0,i}$

$\theta_1 := \theta_1 - \alpha \frac{1}{m} \sum_{i=1}^{m} \big(h_\theta(x_i) - y_i\big) \cdot x_{1,i}$

$\vdots$

$\theta_n := \theta_n - \alpha \frac{1}{m} \sum_{i=1}^{m} \big(h_\theta(x_i) - y_i\big) \cdot x_{n,i}$

4.5 Feature Scaling

Feature scaling is a method to scale the data such that all the weights are in the same range and easily comparable. Gradient descent converges faster when this is applied.

4.5.1 Min-max Normalization

By **rescaling** the range of features to $[0, 1]$ or $[1, -1]$, we can make the weights more comparable. The general formula for a scaled value x' is:

$x' = \frac{x - \min(x)}{\max(x) - \min(x)}$

4.5.2 Mean Normalization

Alternatively, by computing the mean μ_i and standard deviation σ_i for each feature, we can **standardize** the features to have a mean of 0 and a standard deviation of 1. The general formula for a scaled value x' is:

$x' = \frac{x - \mu_i}{\sigma_i}$

4.6 Polynomial Regression

The relationship between features is not always linear.

$h_\theta(x) = \theta_0 + \theta_1 x + \theta_2 x^2 + \theta_3 x^3$ may be valid, and so may $h_\theta(x) = \theta_0 + \theta_1 x + \theta_2 \sqrt{x}$. Feature scaling will need to be done.