CS188 Artificial Intelligence Projects Summary

Niels Joubert

2008-12-05

Abstract

Berkeley's CS188 course teaches the basics of Artificial Intelligence as defined by the **rational agent** approach. Since this course taught a significant amount of material through doing projects, it's worth revisiting what we did and how it worked. All the code is in Python 2.4

Contents

1 Project 1 - Search in Pacman - Agents and Search

1.1 Theory about Searching

Reflex Agents considers only the current precept and memory. It does not consider the future consequences of its actions. **Goal-based Agents** plan ahead, basing decisions on hypothesized consequences of actions.

1.1.1 Graph Searching

Problem consists of:

- State Space - all the current possibilities
- Successor Function - how to advance one state to the next
- Start and Goal - where I am and where I want to get.

Solution is a set of actions that transforms start state to end state. We can't build the whole graph - it's too big. So we find only the important parts. General formulation: Keep a **Fringe** of unvisited nodes and a **Closed Set** of already-visited nodes. Different formulations simply have different **Exploration Strategies**:

- **Depth First Search (DFS)**
  LIFO queue. Always expand the child of the current node first.
  [Complete with cycle checking, Not optimal, time: $O(b^{m+1})$, space: $O(bm)$]
- **Breadth First Search (BFS)**
  FIFO queue. Always expand *all* the successors of a node first.
  [Complete, Possibly Optimal, time: $O(b^{s+1})$, space: $O(b^s)$]
- **Iterative Deepening**
  Combination of BFS and DFS.
  [Complete, Possibly Optimal, time: $O(b^{s+1})$, space: $O(bs)$]
- **Uniform Cost Search**
  Expand node with cheapest cost. BFS if constant costs.[1] Orders by *backward cost*. Priority Queue.
  [Complete, Optimal, time: $O(C^* b^{C^*/\epsilon})$, space: $O(b^{C^*/\epsilon})$]
- **Best First Search / Greedy Search** (Heuristic-based)
  Expand node with lowest heuristic. Best - straight, Worst - whole tree. Orders by *forward cost*.
  [Complete with cycle checking, Not Optimal]
- **A* Search** (Heuristic-based)
  UCS + Greedy. Order by *forward + backward cost*.
  [Complete, Optimal with consistent (admissable for tree) heuristic.]

b is branching factor (amount of children per node)
s is the depth of the shallowest goal
m is the maximum depth of a goal (or, the depth of a goal DFS will fine)
C^* is the cost of the optimal solution (the total path cost).
ϵ is the least cost of an action.

BFS finds the optimal goal if the path length is nondecreasing with depth.
DFS >> **BFS** when there exists many solutions at a similar (possibly deep) depth.
DFS << **BFS** when solutions are sparse.

[1] UCS could be worse than BFS since UCS explores cheapest nodes first, so it might expand significant parts of the tree with small steps.

1.1.2 Heuristics

A heuristic is a function $h(n)$ that maps a *node* to an estimated *cost to goal*. It provides the *forward cost* - an estimate of how expensive it is to reach the goal from the given node. Note that the *backwards cost* is the true cost of the whole path to some node.

Heuristic is **Admissable** if $h(n) \leq h^*(n)$, that is, if the estimated cost is less than the true cost to the nearest goal. Heuristic is **Consistent** if $cost(n, a, n') \geq h(n) - h(n')$, that is, if the true cost of a path exceeds the change in heuristic moving along that path.

Trade off the quality of your estimate with the work needed per node. Thus we try to come up with a **relaxed problem** heuristic: Simplify (relax) the problem specification, and come up with a cost function for the simplified problem.

1.2 Project Overview

This project takes place on a Pacman board with Pacman as the only agent. The project asked us to write two types of search problems - one where you find a path from the start position to a endposition on the board, one where you find a path to each all the food on the pacman board. Both were route-planning problems, but the states in the search problem varied.

1.3 Questions 1 through 4 - Find a Path to a Dot

Files and Functions:
search.py: simpleSearch, informedSearch

We abstracted search into two search functions: Simple Search and Informed Search, both in *search.py*. Simple Search expands nodes one by one and placed it in the appropriate datastructure to generate BFS or DFS. Informed Search included a heuristic to decide how to expand nodes, thus implementing UCS and A* Search.

Nodes in the graph consisted of the (x, y) position of pacman. On the fringe we stored the total path, consisting of a list of (node, the action to get to that node, and the cost of the step from its parent to that node) tuples.

1.4 Questions 5 and 6 - Eating all the Dots

Files and Functions:
search.py: simpleSearch, informedSearch
searchAgents.py: AStarFoodSearchAgent, getFoodHeuristic

A solution is now a path that eats all the dots on the board. A state consists of the whole board - a tuple of (pacman's position, layout of food on grid). Thus the goal state is a grid such that there are no more pellets on the board. Since the state space is now considerably bigger, we need more expansive search techniques.

We had to write an admissible heuristic to run A* on this problem. We also wrote Greedy Search to take advantage of our heuristic. We started off with nothing very fancy, but we could not break our goals. We ended up implementing *Prim's Minimum Spanning Tree Algorithm* as a heuristic.

2 Project 2 - Multi-Agent Pacman - Agents and Game Trees

2.1 Theory about Local and Adversarial Search

Game Trees are a form of local, **adversarial search** problems. **Local Search** is the process of improving what you know until you can't make it any better. This is in general a vast improvement of global search queue-based algorithms discussed for project 1. It is *incomplete* though, and will not guarantee finding any solution.

Hill Climbing is the same as Greedy search, and is the simplest local search algorithm: always choose the best neighbor. This gets stuck in local maxima. **Simulated Annealing**[2] attempts to improve this by allowing downhill moves, but makes them rarer over time (as if it cools down). **Beam Search** is the same as hill climbing, but always keeps at least k states, thus encouraging diversity. **Genetic Algorithms** is similar to beam search, but uses a *natural selection* metaphor to pick neighbors, allowing for *pairwise crossover* with optional mutation.

2.1.1 Adversarial Search for Deterministic Games

Many formalizations are possible. For Example: States, Players, Actions depending on Player and State, and Transition Function $f(S, A) \mapsto S'$. A solution is a **policy**: $p(S) \mapsto A$. We will build a tree. Each node stores a value: the best outcome that can be reached from this node. This is the maximal outcome of its children. Thus, *build the tree bottom up*: construct down to endpoints, calculate endpoint score, propagate score up the tree. This is most easily done *recursively*.

Minimax search alternates players in a **Zero-Sum Game:** Both players use the same score, player A maximizes it, player B minimizes it. Your level of the tree (*a max node*) will choose the action with the highest score. Your opponent's level of the tree (*a min node*) will choose the action with the lowest score.
Time Complexity: $O(b^m)$, **Space Complexity**: $O(bm)$ where b is branching factor and m is depth.
This is too deep, so we *limit the depth*. Loses optimal play guarantee. We need a way to guess the value of the nodes we terminate it.

Evaluation Function: Scores non-terminal states. In practice: Evaluate as a linear sum of features, $Eval(s) = w_1 f_1(s) + \cdots + w_n f_n(s)$. We use a **Feature Extractor** to calculate the value of a feature given some state. These can be *distances*, *amounts left*, *score*, etc.

Alpha-Beta Pruning exploits knowledge about outcomes so far and computational interplay. For Max: α is best option of its Min children so far. Any Min child node of the current node with a child with $score < \alpha$ can be ignored. This works since the Min node will always choose the smallest, so if you've found something smaller on this node's children, then this Min child will be a worse option, so we ignore it. U is the utility.
Zero Sum: $|U_b + U_a| = 0$. Prune when $-U_b \leq \alpha$.
Non-Zero Sum: $|U_b + U_a| \leq \epsilon$. Prune when $-U_b + \epsilon \leq \alpha$.

2.1.2 Adversarial Search for Stochastic Games

If we don't know the result of an action (one of a set of possibilities) we use **Expected Value** as score of node. Thus, introduce *Chance Nodes* that calculated average of children states.[3] This can be formalized as a **MDP**3.

Maximum Expected Utility: Take the action that you believe has the best net response. This is the definition of *rationality*. Expectation $E[f(x)] = \sum_x f(x)\mathbb{P}(x)$. Utilities describe the agent's preference.

[2] Annealing means *to harden something through heat, to make less brittle by heat treatment.*
[3] Notice that we can combine any amount of min, max and chance nodes in a hybrid tree.

2.2 Project Overview

For this project we had to write pacman agents that play against ghosts. Thus, you had one Maximizer player, and two or more Minimizer players. Our agents supported simple reflex agents, minimax agents and expectimax agents.

2.3 Question 1 - Reflex Agent and Move Evaluation Function

Files and Functions:
multiAgents.py: ReflexAgent.getAction(), ReflexAgent.evaluationFunction()

Reflex Agents examine all the actions available according to its successor state with no consideration for the consequences of its actions. Thus, we find all the successor states and evaluate each of them, choosing the highest scoring successor. Our evaluation function was a weighted sum of features:

- Game Score
- Total Dots left
- Distance to closest dots
- Sum of distances to each dot
- Distance to closest ghost

2.4 Question 2 and 3 - Minimax Agent with Alpha-Beta pruning

Files and Functions:
multiAgents.py: MinimaxAgent, AlphaBetaAgent

These were both written as recursive problems, switching between a min and a max cycle in the recursion. Its basically straight from *Artificial Intelligence: A Modern Approach*. MiniMax agents always assume the worst. They play in such a way to try and win even if the opponent plays the perfect game against them.

2.5 Question 4 and 5 - Expectimax Agent and State Evaluation Function

Files and Functions:
multiAgents.py: ExpectimaxAgent, betterEvaluationFunction()

Again, almost straight from the required text. Probabilistic agents concede that the opponent will not play a perfect game against you, and that you can do better by playing in such a way as to maximize your *expected* value, rather than the *minimal* value.

As for our evaluation function, we maximized the score, minimized the amount of dots, moved towards the closest dot, minimize the amount of capsules and take or avoid the appropriate terminal nodes. We came to the conclusion that taking random ghosts into account for any case other than when they're about to kill you gives you poor results.

3 Project 3 - Markov Decision Processes and Reinforcement Learning

3.1 Theory about MDPs and RL

3.2 Project Overview

4 Project 4 - Static Ghostbusters - Bayes Nets

4.1 Theory about Bayes' Nets

4.2 Project Overview

5 Project 5 - Dynamic Ghostbusters - Hidden Markov Models

5.1 Theory about HMMs

5.2 Project Overview

6 Project 6 - Classification - Machine Learning

6.1 Theory about ML

6.2 Project Overview

7 Closing Remarks

CS188 with professor Dan Klein was an incredible experience. Follow-up courses to keep in mind is **CS281A/B** and **CS288**.

References

Artificial Intelligence A Modern Approach. Russel, Norvig.
CS188 Dan Klein lecture slides, Fall 2008.