Finish Adversarial Games, Constraint Satisfaction

CS156

Chris Pollett

Sep 24, 2014

Outline

- Minimax Algorithm
- Alpha-Beta Pruning
- Constraint Satisfaction

Introduction

- On Monday, we started talking about search problems which arise in multi-agent, competitive environments and called them **adversarial search problems** or **games**.
- We gave a formal definition of such games in terms of initial state, PLAYER(s), ACTIONS(s), RESULT(a,s), TERMINAL-TEST(s), and UTILITY(s,p).
- We then focused our attention on two player, zero-sum games with perfect information.
- Starting from the INITIAL_STATE determining actions from states using ACTION(s), and generating resulting states using RESULT(a,s) we can define a game tree for a given game.
- In class, we gave the game tree for tic-tac-toe.
- Finally, last day, we defined the MINIMAX function for a game state which gives the payoff value for a state to a player.
- Today, we use this function to compute what a player should do in a given board situation.

The Minimax Algorithm

The **minimax** algorithm below computes the minimax decision from the current state. So we could use it to actually make an agent that could play a game.

Here $arg\max_{a \in s} f(a)$ returns the element a of set S that has the maximum value of $f(a)$.

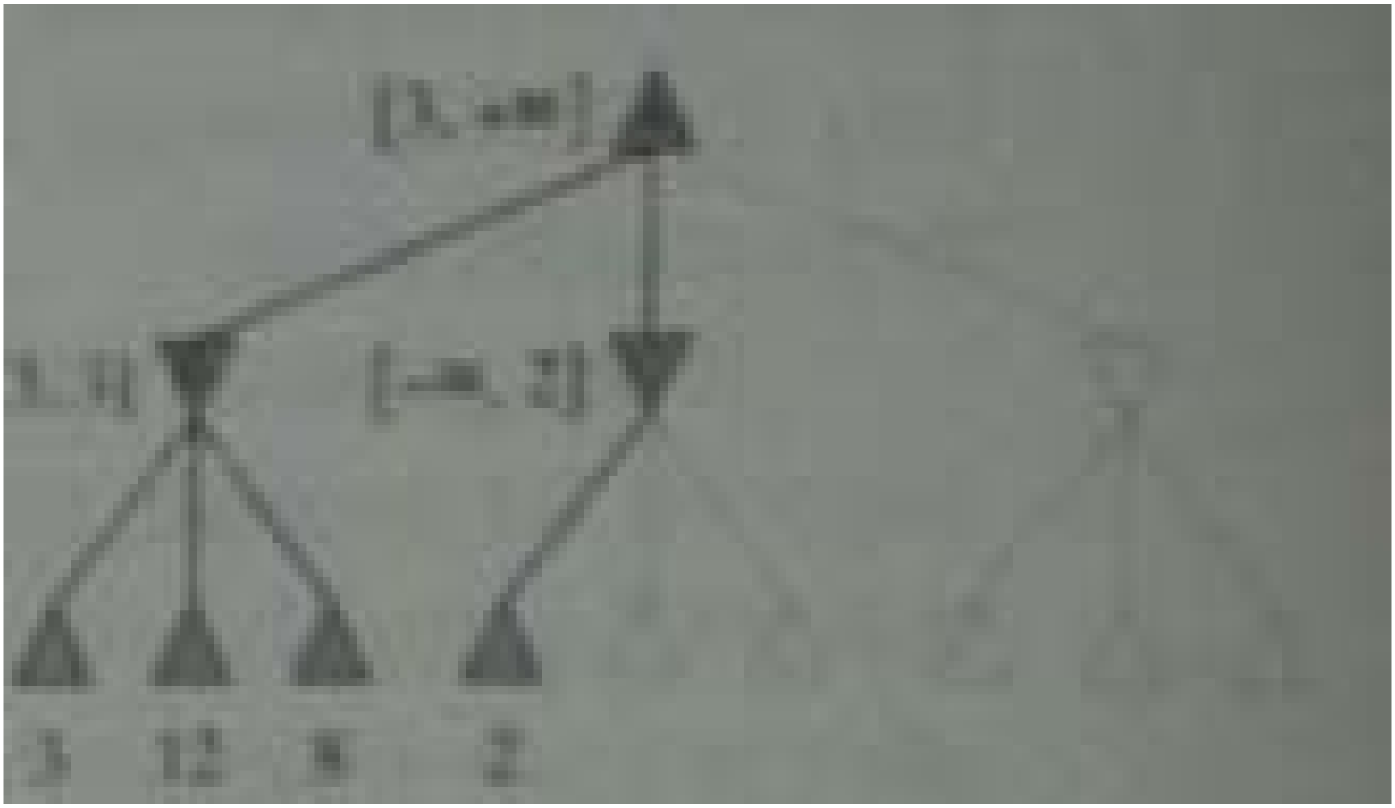
```
function MINIMAX-DECISION(state) return an action //it is assume is MAX's turn
    return argmax_(a in ACTION(s)) MIN-VALUE(RESULT(state, a))

function MAX-VALUE(state) return a utility value
    if (TERMINAL_TEST(state) == true) then return UTILITY(state, MAX)
    v := -infty
    for each a in ACTION(state) do
      v := MAX(v, MIN-VALUE(RESULT(s, a)))
    return v

function MIN-VALUE(state) return a utility value
    if (TERMINAL_TEST(state) == true) then return UTILITY(state,MAX)
    v := infty
    for each a in ACTION(state) do
      v := MIN(v, MAX-VALUE(RESULT(s, a)))
    return v
```

Remarks on Minimax Algorithm

- If the maximum depth of the game tree is m, and expected branching factor is b, then time complexity of minimax is $O(b^m)$.
- It is possible depending on implementation to have a linear space complexity, ergo space complexity is not an issue.
- $O(b^m)$ for time complexity is impractical.
- Can we do better?
- There are a variety of **pruning strategies** that allow a player to quickly determine that a branch is not worth following.
- We look next at one such strategy that can improve the time complexity to $O\left(b^{\frac{m}{2}}\right)$.

Alpha-beta Pruning

- Consider two level tree:



- MAX has a current backed-up value of 3. On the first branch under node C, MIN has a 2, so the largest value MIN could pick is 2 and this is less than 3. So MAX doesn't need to expand the other two nodes under C. The backed-up value 3 is called the alpha value, and ignoring the two remaining branches under C is called doing an **alpha pruning**, or **alpha cut** of the tree.
- The analogous thing for MIN is a **beta value**. And MIN can do **beta pruning** (make **beta cuts**) of tree.
- For MIN, the beta value is the largest value as opposed to alpha's smallest value.
- On average, alpha/beta pruning makes the minimax algorithm time complex $O\left(b^{\frac{m}{2}}\right)$.
- So in the same amount of time one can view a tree twice as deep as straight minimax.
- In order, to achieve this one needs to expand the nodes somehow in close to best first order. For chess, one can get within a factor of two of best first, by considering capture moves before, threat moves, before forward, before backward moves.

Imperfect Real-Time Decisions

- For a game like chess, we can't completely expand out the game tree.
- So we can't determine all the leaf nodes, so how do we do minimax?
- Typically, we have a a heuristic function Eval(s) which gives an estimate for how good a given state is.
- We also have a CUTOFF-TEST(s,d) which returns whether or not to keep evaluating given we are at a depth d and in state s.
- Given these we can define a heuristic minimax as:
  $H-MINIMAX(s,d) :=$
  $$\begin{cases} EVAL(s) & \text{if } CUTOFF-T \\ \max_{a \in ACTION(s)} H-MINIMAX(RESULT(s,a), d+1) & \text{if } PLAYER(s) = \\ \min_{a \in ACTION(s)} H-MINIMAX(RESULT(s,a), d+1) & \text{if } PLAYER(s) = \end{cases}$$

What is a Constraint Satisfaction Problem?

- So far when discussing search, we have looked at environments in the states were indivisible, that is, **atomic**.
- We now consider states which are allowed to have field variables. i.e., we consider environments with factored representations.
- For factored representation, we say a state **solves** the problem when each field variable satisfies all the constraints on that variable.
- A problem described in this way is called a **constraint satisfaction problems,** or CSP.
- Sometimes using a factored representation allows one to eliminate large portions of the search space all at once by identifying variable/value combinations that violate the constraints.

CSP Definition

- A constraint satisfaction problem consists of three components X, D, and C where:
  - X *is a set of variables,* $\{X_1, ...X_n\}$
  - D *is a set of domains,* $\{D_1, ..., D_n\}$ *, one for each variable.*
  - C *is a set of constraints that specify allowable combinations of values.*
- Each domain D consists of a set of allowable values, $\{v_1, ..., v_k\}$ for X_i.
- Each constraint in C consists of a pair $\langle scope, rel \rangle$, where $scope$ is a tuple of variables that participate in the constraint and rel is a relation that those variables can take on.

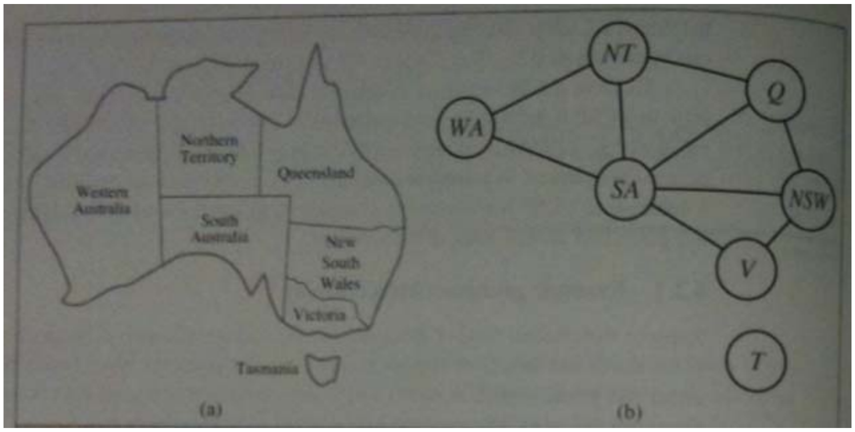
Definition Example

- Suppose $X = \{X_1, X_2\}$ and $D = \{\{A, B\}, \{A, B\}\}$.
- We would like to have the constraint that X_1 and X_2 take different values.
- To do this we could set $C = \{\langle (X_1, X_2), rel \rangle\}$ where rel is the relation $\{(A, B), (B, A)\}$.
- It is often convenient to use common abbreviations for well-known relations.
- I.e., we could write C as $\{\langle (X_1, X_2), X_1 \neq X_2 \rangle\}$.
- Notice the variables themselves are obvious from the relation, so we often abbreviate $\langle (X_1, X_2), X_1 \neq X_2 \rangle$ further as just $X_1 \neq X_2$.

A CSP Solution

- To solve a CSP we need to define a state space and the notion of a solution.
- Each state in a CSP is defined by an **assignment** of values to some or all of the variables, $\{X_i = v_i, X_j = v_j, ...\}$.
- An assignment which does not violate any constraints is called a **consistent** or legal assignment.
- A **complete assignment** is one in which every variable is assigned; an assignment which only assigns values to some of the variables is called a **partial assignment**.
- A **solution** to a CSP is a complete, consistent assignment.

Example: Map Coloring



- Australia consists of seven states and territories. Let's call a state or territory, a region.
- We are given the task of coloring each region red, green, or blue on a map in such a way that no neighboring regions have the same color.
- For this problem, $X = \{WA, NT, Q, NSW, V, SA, T\}$
- The domain D_i for each of these variables is $\{red, green, blue\}$.
- $C = \{SA \neq WA, SA \neq NT, SA \neq Q, SA \neq NW, SA \neq V, WA \neq NT, NT \neq$
- An example solution to the problem might be: $\{WA = red, NT = green, Q = red, NSW = green, V = red, SA = blue, T =$

Remarks

- There exist general-purpose CSP-solving systems. So if you can formulate your program as a CSP, you can just run one of these systems on your problem to get an answer.
- We could have formulated the map problem as a state-space search problem.
- However, in the CSP formulation we can eliminate large portions of the search space more quickly.
- For example, we might start with the empty map, then choose $SA = blue$, due to the constraints, we can conclude immediately that none of the five neighbors can take on the value blue.
- On the other hand a state space searcher would have 3^5 assignments to the neighbors, rather than the reduced 2^5 we get because of this constraint.

Example: Job-shop Scheduling

- Factories have the job of scheduling a day's worth of jobs, subject to various constraints.
- Consider the problem of scheduling the assembly of a car.
- The whole job is composed of tasks, and each task can be modeled as a variable, the value of each variable is the time the task starts, expressed as an integer number of minutes.
- Constraints for job scheduling express things like one task must occur before another task (put wheel on before hubcap), and that certain jobs take a certain amount of time to complete.
- As a concrete example of job scheduling, our variables X might be:
  $$X = \{Axle_F, Axle_B, Wheel_{RF}, Wheel_{LF}, Wheel_{RB}, Wheel_{LB}, Nuts_{RF}, Nuts_{LF}, Nuts_{RB}, Nuts_{LB}, Cap_{RF}, Cap_{LF}, Cap_{RB}, Cap_{LB}, Inspect\}$$
- Next we model the **precedence constraints** between tasks. These are constraints of the form:
  $T_1 + d_1 \leq T_2$
  indicating that T_1 must be done before T_2 and takes at least d_1 time.
- For our example, the precedence constraints look like:
  $$\begin{array}{ll} Axle_F + 10 \leq Wheel_{RF}, & Axle_F + 10 \leq Wheel_{LF} \\ Axle_B + 10 \leq Wheel_{RB}, & Axle_B + 10 \leq Wheel_{LB} \\ Wheel_{RF} + 1 \leq Nuts_{RF}, & Nuts_{RF} + 2 \leq Cap_{RF} \\ Wheel_{LF} + 1 \leq Nuts_{LF}, & Nuts_{LF} + 2 \leq Cap_{LF} \\ Wheel_{RB} + 1 \leq Nuts_{RB}, & Nuts_{RB} + 2 \leq Cap_{RB} \\ Wheel_{LB} + 1 \leq Nuts_{LB}, & Nuts_{LB} + 2 \leq Cap_{LB} \end{array}$$

More Job-shop scheduling

- Suppose we had four workers to install wheels, but they have to share one tool that puts the axle in place.
- We need a **disjunctive constraint** to say $Axle_F$ and $Axle_B$ must not overlap in time; either one come first or the other does:
  $(Axle_F + 10 \leq Axle_B)$ or $(Axle_B + 10 \leq Axle_F)$
- We also might need to assert that the inspection comes last and takes 3 minutes.
- To do this for every variable except Inspect we add a constraint of the form $X + d_X \leq Inspect$.
- As final constraint, we might have the requirement that the whole assembly be done in 30 minutes.
- We can achieve this by limiting the domain of all the variables to: $D_i = \{1, 2, 3, ...27\}$.

Variations on the CSP Formalism

- The simplest kind of CSP variables have **discrete, finite domains**.
- Map-coloring and job-scheduling with time limits are both of this kind.
- The 8-queens problem can be formulated in this way where the variables are $Q_1, ...Q_8$ which range over the domains $D_i = \{1, ..., 8\}$, the position for a given queen.
- A discrete domain can be **infinite**, for example, it might be the integers.
- In such cases, a **constraint language** such as $T_1 + d \leq T_2$ must be used to understand constraint without have to enumerate the set of pairs of allowable values (T_1, T_2).
- There are special solution algorithms for **linear constraints** on integers, but the book doesn't discuss them (look up ellipsoid method).
- The situation where the domain is the integers and the constraints are **nonlinear** can be shown to be undecidable.
- If we take the domains to be an complete, totally-ordered field such as the reals, the domain is said **continuous**.
- Continuous CSPs with linear constraints are known as **linear programming** problems.