Lecture 3: State spaces

AI For Traditional Games
Prof. Nathan Sturtevant
Winter 2013

UNIVERSITY OF DENVER

State Spaces

- What is a state?
  - What does it contain?
  - How many of them are in a game?

Analyze tic-tac-toe

- How many states are possible?
- Several ways to calculate:
  - 9 locations, each of which can have 3 values
    - 19,683 possibilities
    - But, the number of x's and o's should be similar!
  - 9! (362,880) ways to play a game
  - Only 126?(138) ways a game can end
    - Symmetry! (eg Only 3 first moves)

Analyze Nim

- Start with a stack of N items
  - Each person removes either 1 or 2 items
  - The first person who can't remove an item loses
- (Play a game between two students)
- How many states?
  - Only 2*N possible states (why not 2*N+2?)
- How many outcomes?
  - Just 2

Scaling it up?

- What happens if we increased the board size?
- Tic-tac-toe
  - Number of legal moves would greatly increase
  - Number of states would increase
- Nim
  - Number of legal moves stays the same
  - Number of states increases
- Difference between placing pieces and moving pieces

First-level analysis

- How would we build a general model?
  - Assume that there are *b* actions (always the same)
  - Assume we are searching to depth *d*
  - b^d states! (Growing exponentially)
    - Similar implications and algorithms as used in single-agent search

How large are some common games?

- Connect Four has 10^{13}
- Checkers has 10^{20} states; 10^{18} reachable
- Chinese Checkers has 10^{24} states
- Chess has about 10^{47} states
- Go has about 10^{171}

What does this imply?

- If Deep Blue is looking at 300,000,000 states / sec
  - Can't be analyzing anything close to the whole game
- How much memory does our machine have?
  - Can't be storing all these states in memory
  - May want/need an efficient state representation
  - (Allocating memory is very slow)

State Model

- A state is an abstract representation of a game
- A state should provide:
  - A successor function:
    - Get legal moves or legal successor states?
    - Generally use depth-first algorithms and only keep 1 copy of the full game state in memory
    - Moves can often be represented efficiently

State Model

- A state should provide:
  - Functions to apply and undo actions
  - A hash function
    - Will discuss in more depth later
  - Information about whose move it is
  - A test to see if the game is over
  - Information about who won
  - [Ability to copy a state]

Why do we care about this?

- Ken Thompson showed a strong correlation between depth of search and playing strength
  - K. Thompson. “Computer Chess Strength”, Advances in Computer Chess 3, M. Clarke (ed.), pp. 55-56, 1982.
  - That’s why Deep Blue was engineered to explore 300 million positions per second
- How do you improve depth of search?
  - Make your program faster

Example: Hop Step

- 121 board positions
  - Can be blocked or not blocked
- 2 players with 3 pieces each
  - Each in one of 121 positions
  - Lots of actions on pieces, so store piece locations
  - Also need to store parity (1/2) for each piece

UNIVERSITY OF DENVER

Hop Step

- Generating actions (1):
  - Convert index (0…120) into x/y coordinate
  - For each piece, compute offsets to get new x/y coord
  - Convert back into index
- Advantage:
  - General (can change board sizes or movement rule)
- Disadvantage:
  - Slower and harder to implement

UNIVERSITY OF DENVER

Hop Step

- Generating actions (2):
  - For each position on the board
    - Hard code the 1-step and 2-step moves
  - Advantages:
    - End result is fast, avoid extra implementation
  - Disadvantage
    - Lots of work by hand
- Or, try it the first way and cache results

UNIVERSITY OF DENVER

Hop Step

- Once (or as) potential moves have been generated, moves into blocked cells should be disallowed
- Checking to see if game is over:
  - Function will be called a lot
  - Relatively expensive to generate moves and ask if no moves can be made
  - End game on special “null” move

UNIVERSITY OF DENVER

Bugs and Performance

- Test your code to make sure it works (well)!
  - Write a small program that plays out random games
  - Run hundreds or thousands of games
    - Apply and undo moves and verify you’re still at the start state
      - Copy the state and compare that the states are identical
  - Measure the speed and profile the code
  - Save this testing code and run it regularly

Bugs and Performance

- In general you need to prove that your program works in the write-up
  - Particularly important when more complicated algorithms are implemented
  - **Very** easy to introduce bugs
    - This is one reason why I've broken the assignments into small portions
    - Should help you write better code

Bugs and Performance

- Know how to use your data structures well
  - Memory allocation is expensive
    - If you allocate moves on the heap (using new/malloc), you may want to manually cache and re-use moves
      - Harder to do this in Java
  - If you use std::vector<t>, it allocates memory for you
    - Keep as part of your state, so it isn't re-allocated
    - Use carefully externally / pass by reference