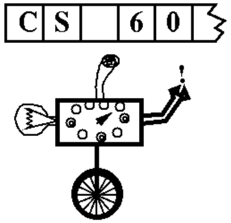


States and Transitions

October 24, 2001

States

- The *state* of a computation in progress is everything we need to know to keep going.
  - Abstract summary of everything that has been done so far
  - What sort of things make up the state of a program?

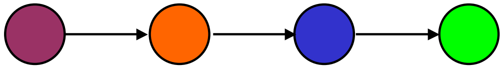
```
int x = 0;
x = x + 3;
x = x * 2;
x = x - 1;
```

Example

- Operating systems allocate the computer's resources among multiple programs
  - Among other tasks
- Multitasking allows many programs to run simultaneously, even on a single CPU
  - Run one program for a short time, then stop and spend some time on the next, etc.
  - At each context switch, OS must save the state of the program so that it can be restarted later.

Transitions

- Typical computations proceed by a series of steps, where each step modifies the state.
  - A step from one state to another is called a *transition*.
- A sequence of transitions can be represented pictorially:



Next States

- Sometimes the current state may not uniquely specify the next state
    - Often, because choice of the transition depends on external information
        - user input, next character in a file, etc.
    - Or, just because of random choice
        - known as nondeterminism
- We can still draw diagrams showing all the possible state transitions.

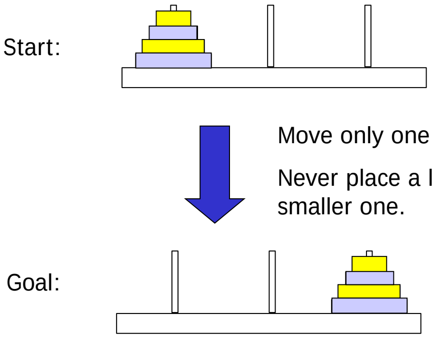
Example

```
int x = 3;
int y = getOneOrTwoFromUser();
x = x * y;
y = y - 1;
```

Puzzles and Games

- Many one and two-player games can be described with states and transitions.
    - The transitions are just the legal moves.
    - What is the state for:
        - nim?
        - peg solitaire?
        - tic-tac-toe?
        - dots and boxes?
        - Traffic Jam?
        - Monopoly?

Example: Towers of Hanoi



Move only one disk at a time.

Never place a larger disk on a smaller one.

How many states are there?

Example State Diagram (2 Disks)

Representing State Diagrams

- The state diagrams for these games can be viewed as directed graphs
  - Nodes are the states
  - Edges are the transitions (legal moves)
- If we allow repetition of states, graph can be made into a *game tree*
  - May be infinite, or at least very large
  - But, avoids having to detect loops

Solving Games

- We might be able to come up with a fixed strategy (e.g., nim)
- Otherwise, we can always try searching the state diagram.
  - For one-player games like the Towers of Hanoi, a solution would be a path from the initial state to the final state.
  - For two-player games, situation is more complex.

Searching Graphs

- Generally, we don't need to first generate the tree or graph and then search it;
  - Can be generated on demand.
  - Given the current state, figure out what the following states are and proceed.

- For the moment, assume we are interested in game trees.

Depth-First Search for Nim

- Recall: depth-first searches a node by first searching everything starting with the first child, and only then going on to the next child
- Start with a pile of 2 and a pile of 1.

Breadth-First Search for Nim

- Recall: depth-first searches a node by first searching everything starting with the first child, and only then going on to the next child
- Start with a pile of 2 and a pile of 1.

DFS vs. BFS

- Depth-first search
  - Is often easier to implement
  - Usually requires less memory
    - The state of the depth-first-search algorithm is just the path from the start to the node currently being searched.
  - But, if there are multiple solutions, may not find the best one.
  - Also, if we don't check for loops we might never find a solution.
    - BFS is guaranteed to eventually find the shortest solution path, if one exists.

Implementing Basic DFS

`DFS(Node) =`

If `Node` is our goal, then we're done.

Otherwise, compute the children of Node.

Call `DFS` on the first child; if this fails go on to the next, ..., until we run out of children.

Implementing Basic BFS

```
BFS(Queue) =
   Dequeue to get Node
   If Node is our goal, we're done.
   Otherwise, enqueue all the children of Node
   and repeat.
```

Avoiding Loops

- To avoid getting stuck in loops we can keep track of where we've been; never search a node we've already seen.
  - May be necessary to detect *lack* of solution.

- Costs:
  - Time to figure out whether we've seen the current state or not.
  - Memory to record all the states we've seen so far.

Other Search Strategies

- Iterative Deepening
  - Variant of DFS where we bound on how deep to search
  - If we don't find a solution, increase the bound & try again.
- A* search
  - Estimate for each state how close it is to a solution
  - At each step look at the node which appears to lead to the shortest solution.
- Alpha-beta pruning
  - For two-player games.
  - Allows portions of the game tree to be skipped as obviously suboptimal

- To learn more about search algorithms, take AI course

Solving the Towers of Hanoi

- One of the following two recursive ideas works:
  - Move the top disk out of the way, recursively move the remaining n-1 disks to the destination, then move the top disk back.
  - Recursively move the top n-1 disks out of the way, move the bottom disk to the destination, and then recursively move the top n-1 disks to the destination.