States and Transitions

What is a “State”?

- A **state** is an abstraction of all relevant aspects of a situation at a given point in time, or in a given position in a sequence.

What is a “Transition”?

- A **transition** represents the change in state between two successive points in time, or between two successive positions in a sequence.

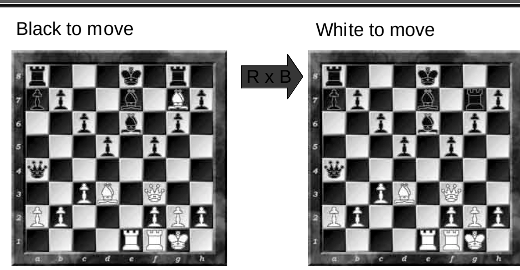
Why is this Important?

- Many computational models are expressible using the ideas of states and transitions.
- So are many aspects of problem solving, game playing, etc.

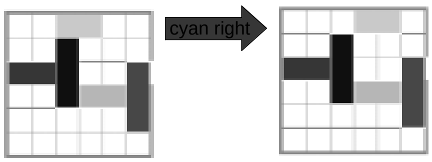
States of a Program in Execution

- A program in execution can be represented by transitions from one state to the next.
- A state consists of:
  - For each variable in the program:
    - the current value of that variable
  - The position of the “instruction pointer” in the program.
- We already demonstrated this in the discussion of McCarthy’s transformation.

States and Transition of a Game



States and Transition of a Puzzle

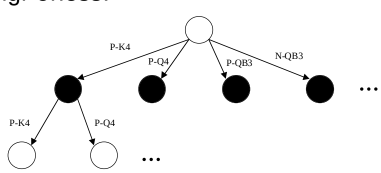


States and Transitions as a Directed Graph

- Nodes = States
- Arcs = Transitions
- Show all possible moves, not just a single sequence
- Usually not constructed explicitly
  - too big

Partial Construction

- Start from an initial state
- Construct "tree" of possibilities
- e.g. Chess:

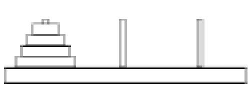


Not Necessarily a Tree

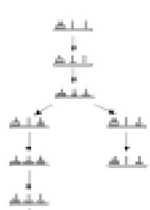
- Fan-in is possible
- Moves can be "commutative":
  - a b = ba
- Completely different ways to achieve same net effect:
  - a b c d e f g = h i

Towers of Hanoi Puzzle

Physical setup

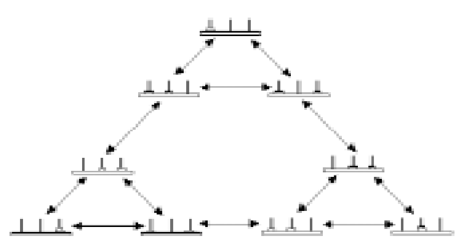


Partial tree



Towers of Hanoi Puzzle

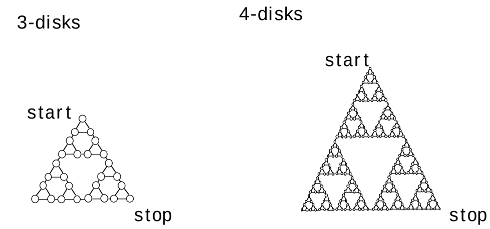
Complete graph for 2-disks



Towers of Hanoi Combinatorics

- Adding one more disk triples the number of states:
  - For each state of an n-disk system, the n+1th can be placed at the bottom of any of the 3 stacks.
  - Therefore 3^n states for n-disk system.

Why is the puzzle "hard"?



Ways to Solve Puzzles

- Algorithmic
- Search
- Heuristic search

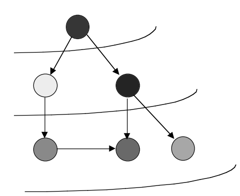
Shortest Solution to Puzzle

- Assuming search:
  - Breadth-first
  - Iterative deepening
    - Successive depth-first searches with a depth cutoff

Breadth-First Search Revisited

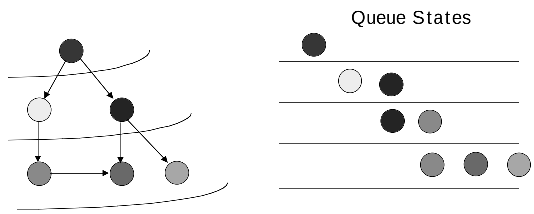
- Déjà vu:
  - Chapter 4 (graph searching)
  - Lecture notes (4th set of slides)
  - Mid-term, problem 2

View as a "Wavefront"



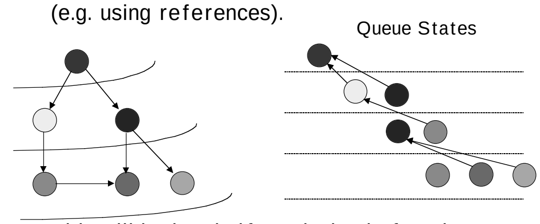
What Data Structure?

- Use a Queue for Breadth-First



Finding your Way

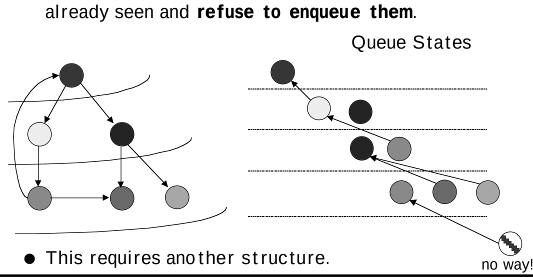
- Have states remember their predecessors (e.g. using references).



- This will be handy if a solution is found.

Termination

- To ensure termination, must **remember** states already seen and **refuse to enqueue them**.



- This requires another structure.

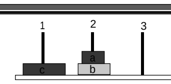
Structures for Loop Detection

- Simple linked list
  - Possibly an extension of your queue
- Hashtable
  - Array of linked lists
  - Much faster

State Representation

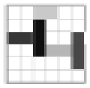
- State representation does not have to be **literal**.
- Any structure that is sufficient to capture all relevant information will do.
- Often there are many choices.
- The choice may impact search efficiency.

Example of State Representation Choices: Towers of Hanoi



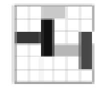
- Mapping from disk to spindle number:
  - ((a 2) (b 2) (c 1))
- List of disks on each spindle, smallest first:
  - ((c) (a b) ( ))
- Solving using rex, for example, the second would probably be more efficient:
  - All manipulation takes place at the tops of the stacks

Example of State Representation Choices: Traffic Jam



- The fact that we already have a representation influences.
- Probably best *not* to replicate entire grid structure as state:
  - Heavyweight states
- I used array of **offsets** for states:
  - Each offset indicates motion of a car from the original state.

Example of State Representation Choices: Traffic Jam



- States = arrays of offsets:
  - initial state = [0, 0, 0, 0, 0]
    understood: [red, blue, cyan, green, magenta] by position
  - immediate transitions to:
    [0, 0, 1, 0, 0] (move cyan right)
    [0, 0, -1, 0, 0] (move cyan left)
    [0, 1, 0, 0, 0] (move blue down)
    [0, 0, 0, 0, 1] (move magenta down)
    [0, 0, 0, 0, -1] (move magenta up)
- From each array, and the initial grid, we can construct the resulting grid.

States in Computing Machines

- Broad split:
  - Finite-state systems
    - Finite-state machines (chapter 12)
    - Finite-state systems with a very large number of states (most computers)
  - Infinite-state systems:
    - Turing machines
    - Most programming languages
    - Other structured automata

Turing Machines

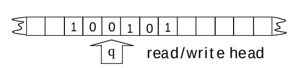
- A very primitive model of computing
  - Conceptually simple to construct
- A very powerful model of computing
  - Any conceptually computable function can be computed by some Turing machine.

Alan M. Turing

- Celebrated mathematician/computer scientist, 1912-1954
- Code-breaking machine
  (play: "Breaking the Code")
- Reaction-diffusion equations
- Proving programs correct
- Artificial Intelligence
- Theory of computability

Turing Machine Depiction

infinitely-extendable tape (blank cells)



(finite alphabet)

read/write head

control state (one of a finite set)

Control is determined by a finite set of **rules** (each a 5-tuple):
(current-control-state, current-read-symbol, next-control-state, write-symbol, head-motion)
with the last three components being functions of the first two.

tm program on turing

- /cs/cs60/bin/tm
- Examples in: /cs/cs60/tm/*.tm
- Sample execution:

Turing machine simulator
rule set

```
turing > tm add1.tm
 1 0 1 1 1 1 1 1        input I typed
 1 1 0 0 0 0 0 0 [_]    output (final tape)
end                     final control state
```

Rule set add1.tm

(current-control-state, current-read-symbol, next-control-state, write-symbol, head-motion)

```
start   _   _   left    add1
add1    0   1   right   end
add1    _   1   right   end
add1    1   0   left    add1
end     0   0   right   end
end     1   1   right   end
```

Assumes that head is to the right of the non-blank symbols on the tape.

Trace of add1.tm (use -t flag)

```
 1 0 1 1 1 1 1 1 [_]
start
 1 0 1 1 1 1 1 [1]  _
add1
 1 0 1 1 1 1 [1]  0 _
add1
 1 0 1 1 1 [1]  0 0 _
add1
 1 0 1 1 [1]  0 0 0 _
add1
 1 0 1 [1]  0 0 0 0 _
add1
 1 0 [1]  0 0 0 0 0 _
add1
 1 [0]  0 0 0 0 0 0 _
add1
 1 1 [0]  0 0 0 0 0 _
end
 1 1 0 [0]  0 0 0 0 _
end
 1 1 0 0 [0]  0 0 0 _
end
 1 1 0 0 0 [0]  0 0 _
end
 1 1 0 0 0 0 [0]  0 _
end
 1 1 0 0 0 0 0 [0]  _
end
 1 1 0 0 0 0 0 0 [_]
end
```

Other Examples

- Complete binary adder: see add.tm
- Busy Beaver n: (How many 1's using only n rules).
- TM simulating TM

Turing's Thesis

- Any computable function is computable by some Turing machine.
- Generally accepted.
- Cannot be proved.

Implications

- The set of all computable functions can be enumerated (there is a "countable" number of them).
- There are non-computable functions.
- There are problems of interest that are unsolvable.

Non-Proof of Turing's Thesis

- Proving Turing's thesis formally would require **formalizing** the notion of computability.
  - This is what Turing set out to do.
  - But that formalization could be argued and to prove or disprove it would require a proof that ***that*** formalization completely captured the notion of computability.
  - We'd then be in a position similar to proving Turing's thesis.

Disproving Turing's Thesis

- Conceptually this is possible:
  - Find a function that everyone agrees is computable.
  - Prove that no TM can compute it.
- However, it is highly unlikely.
- All attempts to characterize computability have been shown to be equivalent to the TM.

An Unsolvable Problem

- Consider any reasonably rich programming language (such as rex or Java).
- Any computable function can be computed in such a language (given sufficient memory).
- Each program in the language is a finite string of symbols.
- We can enumerate the set of syntactically-correct programs.

An Unsolvable Problem

- Enumerate the programs: P_0, P_1, P_2, ...
- The possible inputs to those programs are just strings of characters. They can be enumerated too: I_0, I_1, I_2, ...
- A reasonable question to ask is:

  Does a program P halt on input I, or not?

  This question is "**undecidable**".

Undecidability

- "Undecidable" means there is no algorithm (i.e. program) that can compute the answer.
- General question:

  Does P_i halt on I_j?
- This is a "2-input" question $P_i(I_j)$.

Undecidability

- "2-input" question: Does $P_i(I_j)$ halt?
- Let's simplify this to a 1-input question:

  Does $P_i(P_i)$ halt?

  where the argument means the source code of P_i.
- If we can solve the 2-input question, we can solve the one input one, by just copying P_i for both the program and the input of the original.
- We could then also solve this related problem:

  Does $P_i(P_i)$ diverge? (diverge = not halt)

Undecidability

- Does $P_i(P_i)$ diverge?
- Suppose this were decidable.
- Then there would be a program P_k which computes the answer for *any* input P_i.
- But then we have a slight conundrum:
  - $P_k(P_k)$ gives an answer indicating whether $P_k(P_k)$ diverges.
  - We'd be forced to conclude that $P_k(P_k)$ does not diverge; if it diverged, we'd have a contradiction.

Undecidability

- To really prove the point, we need a slight modification of the function to be computed: With input P_i:
  - Return "yes" if $P_i(P_i)$ diverges.
  - Intentionally diverge if $P_i(P_i)$ does not diverge.
- If the original function were computable, so would this one be.
- Now suppose that *this* function is computable by a program P_k.

The "Halting Problem" is Undecidable

- Now we have an unavoidable contradiction:
  - $P_k(P_k)$ returns "yes" if $P_k(P_k)$ diverges.
  - $P_k(P_k)$ diverges if $P_k(P_k)$ does *not* diverge.
- We are forced to conclude that the original function, determining whether a program halts on an arbitrary input, is not computable after all.

Esoteric Problems?

- The halting problem, despite its practical attractiveness, may seem far removed from your computing experiences.
- However, many other "every day" problems can be shown to be non-computable.
- This is done by showing that within those problems lies the power to simulate a Turing machine.
- With such power comes inherent limitations.

Post's Correspondence Problem

- Is there an algorithm that will accept any finite collection of pairs of strings $[x_1, y_1]$ $[x_2, y_2]$ ... $[x_n, y_n]$ and determine whether there is a "correspondence"

  $x_{i1}\ x_{i2}\ x_{i3} \dots x_{in} == y_{i1}\ y_{i2}\ y_{i3} \dots y_{in}$

  (pairs are allowed to be used multiple times in achieving a correspondence)?
- Example: [a, aaa], [abaaa, ab], [ab, b]
  Correspondence: abaaa a a ab == ab aaa aaa b

An Abstract Generalization

- **Rice's Theorem:**
  Any non-trivial property of computable functions is undecidable on the set of representations of those functions.
- (Non-trivial means not uniformly true or false for all functions.)
- This theorem arises in the study of "Recursion Theory" or "Theory of Computation".