A little perspective

“NP-completeness is not a death certificate – it is only the beginning of a fascinating adventure.”

-- Dasgupta et al. (our textbook authors)

Objectives

- Formulate algorithmic solutions to (intractable) problems
- Use a systematic approach called “state-space search”
- Start with brute-force back-tracking state-space search
- Improvements: intelligent back-tracking state-space search

Graph Coloring Problem

- Assign colors to the vertices of a graph
  - So that no adjacent vertices share the same color
  - Vertices i and j are adjacent if there is an edge from vertex i to vertex j.
- Find all m-colorings of a graph
  - i.e., find all ways to color a graph with at most m colors.
State-Space Search

- What to store in each state?
- What to store in the initial state?
- How to expand a state into successor states?
  - i.e., How to generate children?
- How to search?
- Properties of the state space:
  - Is it a graph with shared sub-structure?
  - Is it simply a tree?

Example: Graph Coloring

**m = 3 colors**

**Initial State:**
- Copy of graph
- No colors
- Mark on one vertex to indicate which vertex to color first

**State:**
- Copy of graph
- Colors assigned to a subset of the vertices
- Mark to indicate which vertex to color next

**State Expansion:**
For all possible updates to the marked vertex:
- Copy parent state
- Assign a color to the marked vertex
- Advance the mark to another vertex

State-Space Search Options

1. **Brute-force back-tracking state-space search**
   - No pruning
   - Pure enumeration
   - Let’s start with this idea.

2. **Intelligent back-tracking state-space search**
   - Pruning
   - i.e., Test partial states and eliminate those that are not feasible
Example: Graph Coloring

Algorithm: Brute-force State-space Search

```python
procedure brute-force-explore(state s)
    Input: current state s
    Output: none; prints solutions
    for each \( s' \in \text{successors}(s) \)
        if criterion(\( s' \)) then
            output "Solution: \( s' \)
        else // possible solution not yet fully specified
            explore(\( s' \))
    Outer-most call: \( \text{explore( init_state() )} \)
```

For the \( m \)-coloring example, we put a little bit of intelligence in the \( \text{successors()} \) function.
Aside: Coloring a Map

- Assign colors to countries so that no two bordering regions are the same color.
- Map coloring is reducible to planar graph coloring.

Color Theorem

- How many colors do you need for a planar map?
- First proposed in 1852
- Answer: Four
- Proved by Haken and Appel in 1976
  - Used a computer program
  - Checked 1,476 graphs, i.e., cases in the proof
  - Required 1,200 hours of run-time

State-space size

- Consider how large the set of leaf states in the search space could be
  - (without pruning)

$n$-Queens Problem

- Given: $n$-queens and an $n \times n$ chess board
- Find: A way to place all $n$ queens on the board s.t. no queens are attacking another queen.

- Can you formulate an algorithmic solution to this problem as state-space search?
- How would you represent a state?

First State-Space Idea

- Consider all possible placements of the queens on a chess board.
- Represent a state as a chess board.
  - How many placements are there?

First State-Space Idea

- Consider all possible placements of the queens on a chess board.
- Represent a state as a chess board.
  - How many placements are there?
  \[
  \binom{n^2}{n} = \binom{64}{8} = 4,426,165,368
  \]
  - Recall: \[
  \binom{n}{k} = \frac{n!}{k!(n-k)!}
  \]
Second State-Space Idea

- Don't place 2 queens in the same row.

Now how many positions must be checked?

Represent a positioning as an ordered tuple \((x_1, ..., x_8)\)

Where each element is an integer 1, ..., 8.

\[ n^8 = 8^8 = 16,777,216 \]
Third State-Space Idea

- Don’t place 2 queens in the same row or in the same column.

Don’t place 2 queens in the same row or in the same column.

Represent a state as a permutation of (1, 2…8)

Now how many positions must be checked?

It is easy to enumerate all permutations

We went from \( C(n^2, n) \) to \( n^n \) to \( n! \)

We applied explicit constraints in the enumeration of successor states to shrink our state space up front.

\( n! = 40,320 \)
Lessons

- Be smart about how to represent states.
- So far, efficiency of state-space search depends on:
  1. The number of successor states satisfying the explicit constraints
  2. The time to generate the successor states

Intelligent Back-tracking
State Space Search

- Aside from intelligently describing states,
- Do: We check a solution after placing all queens:
  - criterion function (i.e., a solution test): yes or no
- To do: We should check after each placement of a queen.
  - partial-criterion function (i.e., a feasibility test): yes or no
  - Let’s see how this will help …

Four Queens Problem

Easier to manage in a lecture.
Let’s illustrate in more detail how intelligent back-tracking works.

Each branch of the tree represents a decision to place a queen.
The criterion function can only be applied to leaf states.

Is this leaf state a solution?
Using only a criterion function (at the leaves), we would build out the entire state space.

A better question: Is any child of this state ever going to be a solution? (before placing queens #3 and #4)

The partial criterion or feasibility function checks whether a state may eventually have a solution.

Will this state have a solution? No. So we prune and backtrack early.

Will this state have a solution? Maybe.
Four Queens Problem

Will this state have a solution? No. Etc.

And so on …

Using the feasibility function, the resulting state space is shown in dark blue.

And so on …

Algorithm: Intelligent Back-tracking
State-space Search

procedure intelligent-explore(state s)
  Input: current state s
  Output: Solution or {} (failure)

  for each s’ ∈ successors(s)
    if criterion(s’) then
      output “Solution:” s’
    else if partial_criterion(s’) then
      explore(s’)
    else
      discard s’

Outer-most call: explore( init_state() )

Efficiency of Back-tracking
Algorithms depends on:

1. The number of successor states satisfying the explicit constraints
2. The time to generate the successor states
3. The proportion of successor states satisfying the partial criterion function
4. The time to compute the partial criterion (feasibility)
   - Can substantially reduce the number of states that are generated.
   - But: also takes time to evaluate.
Conclusions

- Can formulate an algorithm for just about any problem as a state-space search.
- Have some intelligent options for implementing state-space search
- State spaces are large for the hard problems
- But some state spaces are small
  - e.g., for the combinatoric algorithms encountered so far in the course

Assignment

- Homework #23: State-space search
- Read Sec. 9.1.2