Class Meeting #8

8 Puzzle

COS 226

Based in part ok slides by Jérémie Lumbroso and Kevin Wayne
Level-order traversal of a binary tree.

- Process root.
- Process children of root, from left to right.
- Process grandchildren of root, from left to right.
- ...

level-order traversal: \text{SETARCHM}
Q1. Given binary tree, how to compute level-order traversal?

```java
queue.enqueue(root);
while (!queue.isEmpty()) {
    Node x = queue.dequeue();
    if (x == null) continue;
    StdOut.println(x.item);
    queue.enqueue(x.left);
    queue.enqueue(x.right);
}
```

level-order traversal: S E T A R C H M
Q2. Given the level-order traversal of a BST, how to (uniquely) reconstruct?

Ex. $S E T A R C H M$
EVEN-DRIVEN SIMULATION DEMO
KEY INGREDIENTS!
What is a graph?

Like a binary tree, except there can be cycles.
What is a Priority Queue?

- Comes in two flavors: MinPQ / MaxPQ

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MinPQ()</td>
<td>create an empty priority queue</td>
</tr>
<tr>
<td>MaxPQ()</td>
<td>create a priority queue with given keys</td>
</tr>
<tr>
<td>void insert(Key v)</td>
<td>insert a key into the priority queue</td>
</tr>
<tr>
<td>Key delMax()</td>
<td>return and remove the largest key</td>
</tr>
<tr>
<td>delMin()</td>
<td></td>
</tr>
<tr>
<td>boolean isEmpty()</td>
<td>is the priority queue empty?</td>
</tr>
<tr>
<td>Key max()</td>
<td>return the largest key</td>
</tr>
<tr>
<td>min()</td>
<td></td>
</tr>
<tr>
<td>int size()</td>
<td>number of entries in the priority queue</td>
</tr>
</tbody>
</table>
What is a Board?

- Immutable type (defensive copy)
- Knows how to compute neighbors
- Estimates how far from goal

```
public class Board {
    public Board(int[][] tiles)  // construct a board from an N-by-N array of tiles
                                // (where tiles[i][j] = tile at row i, column j)
    public int tileAt(int i, int j)  // return tile at row i, column j (or 0 if blank)
    public int size()  // board size N
    public int hamming()  // number of tiles out of place
    public int manhattan()  // sum of Manhattan distances between tiles and goal
    public boolean isGoal()  // is this board the goal board?
    public boolean isSolvable()  // is this board solvable?
    public boolean equals(Object y)  // does this board equal y?
    public Iterable<Board> neighbors()  // all neighboring boards
    public String toString()  // string representation of this board

    public static void main(String[] args) // unit testing (required)
}
```
WHAT IS A* SEARCH?
Example run

- Solve problem for board on left
- Draw graph of all boards
- Schematize search through graph of boards
- Show role of MinPQ

- This is `puzzle04.txt`
Boards on the priority queue

Board not reachable from initial board
Boards on the priority queue
PROBLEM SOLVED!
```plaintext
% more puzzle04.txt
3
0 1 3
4 2 5
7 8 6

% java-algs4 Solver puzzle04.txt
Minimum number of moves = 4
3
0 1 3
4 2 5
7 8 6

3
1 2 3
4 0 5
7 8 6

3
1 2 3
4 5 0
7 8 6

3
1 2 3
4 5 6
7 8 0
```

Observations

• The graph is MUCH TOO BIG
• Some boards are not reachable from start

(carefully read part about “unsolvable puzzles”)

• The MinPQ (Priority Queue) always contains a fringe of boards that we should look at next
A* search

• Use a “priority function” to try to guide the search through the large graph
• Some conditions on this priority function, but basically
  \[ \text{priority} = \text{estimated min. number of moves} \]
• We give:
  – Hamming (number of misplaced squares + moves so far)
  – Manhattan (sum of distances to correct position)
  – other ideas?
TIPS
Tip #1: Avoid Dropbox Timeout

• Too much (Terminal) output
  – remove print out statements
  – or use assert / debugging that can be turned off easily

• Infinite loops

• Much more memory usage than predicted
  – it may be useful to test only one file at a time in Dropbox
Tip #2: Board before Solver

• Fully test Board.java before doing Solver.java

• If Board.java is not fully tested, things can go very very very very wrong in Solver.java
Tip #3: Iterable neighbors

• You have to implement:

```java
// return the neighboring board positions,
// as an Iterable
public Iterable<Board> neighbors() {
    ...
}
```

• **Idea**: create a Queue (or Stack), add boards to it, and return Queue (or Stack)

• Queue/Stack are Iterable objects
Tip #4: Class SearchNode

- In Solver.java, create a SearchNode
- This [immutable] SearchNode will wrap around a Board, and make it **Comparable** (by priority)
- Being **Comparable** is needed to use MinPQ

```java
private static class SearchNode implements Comparable<SearchNode> {
    // ...
}
```

- SearchNode should also have a pointer to the previous Node (so you can remember the solution)
Tip #5: Test Equality of Board

- The **critical optimization** is making sure we don't go back and forth between two boards (may cause infinite loop, or significantly delay search)
- To avoid this, **Board needs to implement equals**
- Tricky!
Equality test

All Java classes inherit a method `equals()`.

**Java requirements.** For any references `x`, `y` and `z`:

- Reflexive: `x.equals(x)` is true.
- Symmetric: `x.equals(y)` iff `y.equals(x)`.
- Transitive: if `x.equals(y)` and `y.equals(z)`, then `x.equals(z)`.
- Non-null: `x.equals(null)` is false.

**Default implementation.** `(x == y)`

**Customized implementations.** `Integer`, `Double`, `String`, `java.io.File`, ...

**User-defined implementations.** Some care needed.
Implementing equals for user-defined types

Seems easy.

```java
public class Date implements Comparable<Date>
{
    private final int month;
    private final int day;
    private final int year;
    ...

    public boolean equals(Date that)
    {
        if (this.day != that.day) return false;
        if (this.month != that.month) return false;
        if (this.year != that.year) return false;
        return true;
    }
}
```

check that all significant fields are the same
Implementing equals for user-defined types

Seems easy, but requires some care.

```java
public final class Date implements Comparable<Date> {
    private final int month;
    private final int day;
    private final int year;
    ...

    public boolean equals(Object y) {
        if (y == this) return true;
        if (y == null) return false;
        if (y.getClass() != this.getClass())
            return false;
        Date that = (Date)y;
        return (this.day != that.day ) ||
               (this.month != that.month) ||
               (this.year  != that.year );
    }
}
```

typically unsafe to use equals() with inheritance
(would violate symmetry)

must be Object.

optimize for true object equality

check for null

objects must be in the same class
(religion: getClass() vs. instanceof)

cast is guaranteed to succeed

check that all significant
fields are the same
Equals design

"Standard" recipe for user-defined types.
• Optimization for reference equality.
• Check against null.
• Check that two objects are of the same type; cast.
• Compare each significant field:
  – if field is a primitive type, use == but use Double.compare() with double
    (to deal with -0.0 and NaN)
  – if field is an object, use equals() apply rule recursively
  – if field is an array, apply to each entry can use Arrays.deepEquals(a, b)
    but not a.equals(b)

Best practices.
• No need to use calculated fields that depend on other fields.
• Compare fields mostly likely to differ first.
• Make compareTo() consistent with equals().
  x.equals(y) if and only if (x.compareTo(y) == 0)
  e.g., cached Manhattan distance
Two optimizations

• **Critical**: Avoid adding the neighbor “you just arrived from” to the priority queue:

• Cache Manhattan distance inside the board as an instance variable and compute in the constructor (to avoid recomputing it)
When is a board solvable?

When is a board solvable?
When is a board solvable?

• An odd-size board is solvable if and only if the number of inversions is even.
• If $n$ is even, the board is solvable if and only if the number of inversion plus the row of the blank square (counting from 0) is odd.
Click OK and wait. Check will be used even if you choose cancel!