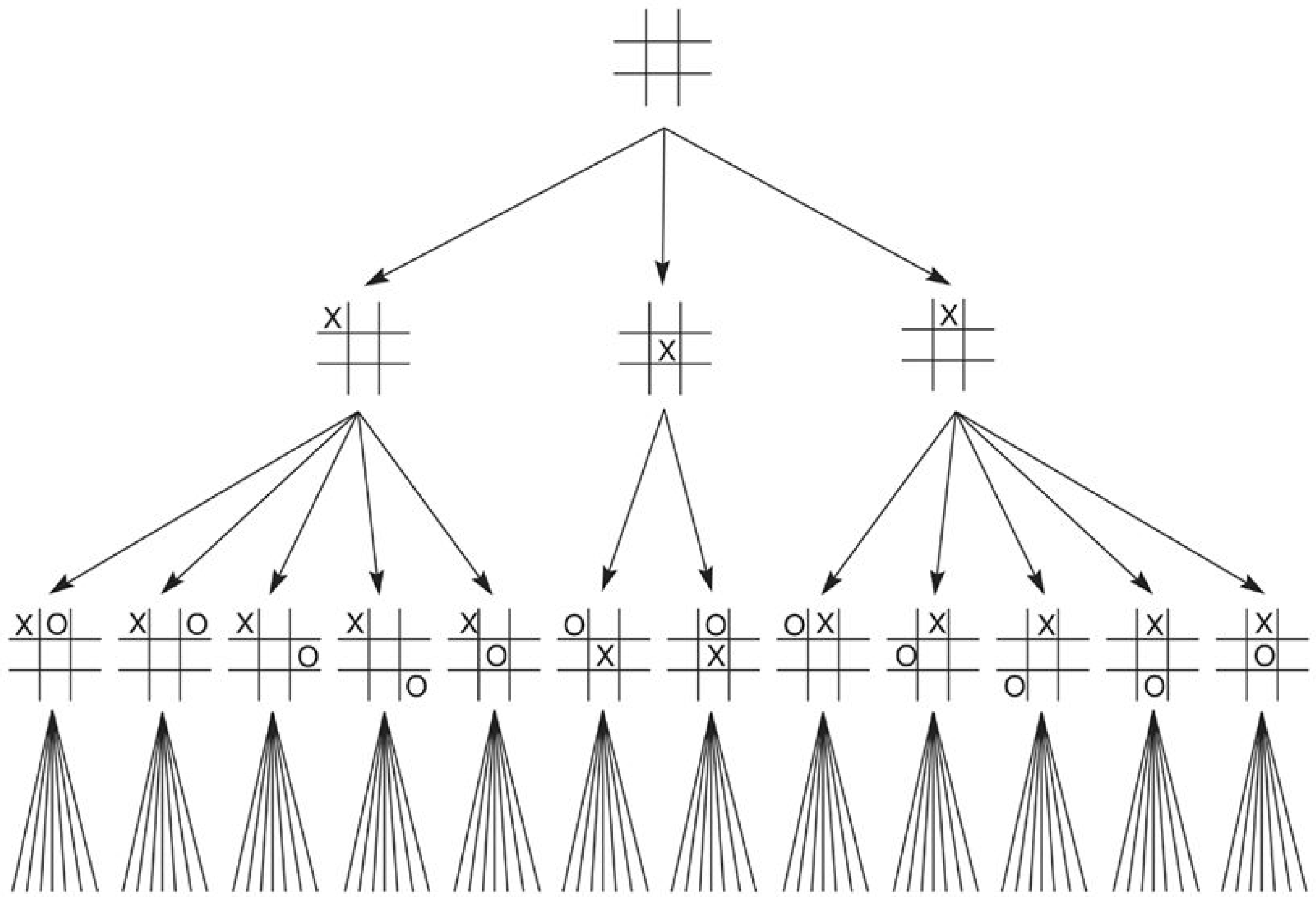
Sliding games: Collections, Generics

Tutorial 11 (28th April 2021)

State-Space Search

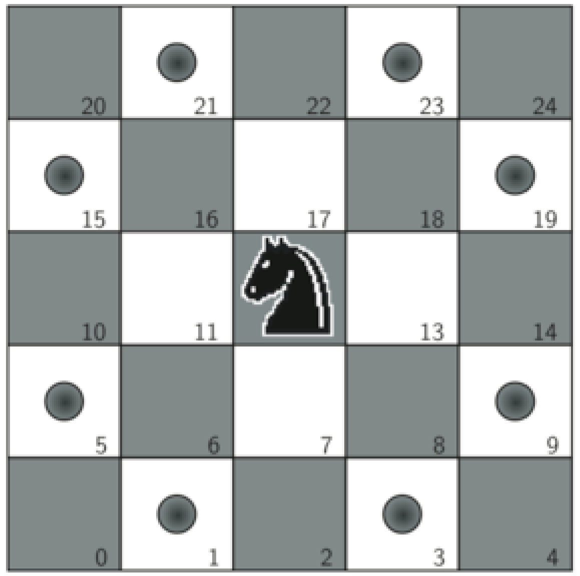
- Many problems take the form of state-space search.
- The states might be legal board configurations in a game, towns and cities in some sort of route map, collections of mathematical propositions, etc.
- The state-space is the collection of the possible states and how they connect to each other e.g. the legal moves between states.
- When we don't have an ***algorithm*** which tells us definitively how to negotiate the state-space we need to search the state-space to find an optimal path from a start state to a goal state.
- We can only decide what to do (or where to go), by considering the possible moves from the current state, and trying to look ahead as far as possible. Chess, for example, is a very difficult state-space search problem.

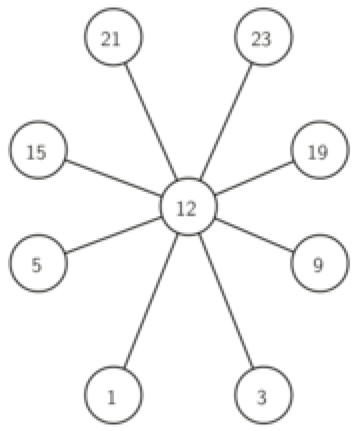
First three levels of the tic-tac-toe state space reduced by symmetry.



Depth-First Search: Knight tour

- Depth-first search: Follows the left-most branch of the search tree first; following it down until it either finds the goal state or hits a dead-end. It will then backtrack to find another branch to follow.





Breadth-First Search: Sliding game

- A simple, common alternative to depth-first search is: breadth-first search.
- Checks every node at one level of the space, before moving onto the next level.

Configuration (state) interface

```
public interface Configuration extends Comparable<Configuration> {
    public abstract Configuration parent();

    public abstract Collection<Configuration> successors();

    public abstract boolean isSolution();

    public default List<Configuration> pathFromRoot(){
        throw new UnsupportedOperationException(
            "pathFromRoot: not supported yet." );
    }
}
```

The interface `Collection`

- we have several containers in Java
  - `String, StringBuffer, ArrayList, Vector, Set`
- many similar operations on these containers
  - `isEmpty, contains, equals, size`
- the interface `Collection` yields a uniform way to handle these kind of operations

The interface Comparable

```
public interface Comparable<T> {
   int compareTo( T other );
}
```

- The relational operator > cannot be used with reference types, but it's possible to compare two objects of the same class if that class implements `Comparable`
- `Comparable<T>` objects have a `compareTo` method.
  - The method *must* return 0 if the objects are equal, a negative integer if **`this`** is less than `object` or a positive integer if **`this`** is greater than `object`.
- A benefit of implementing interface `Comparable<T>` is that `Comparable<T>` objects can be used with the sorting and searching methods of class `Collections` (package `java.util`).

Generic maximum

```
// determines the largest of three Comparable objects

public static <T extends Comparable<T>> T maximum( T x, T y, T z ) {
    T max = x; // assume x is initially the largest

    if ( y.compareTo(max) > 0 )
      max = y; // y is the largest so far

    if ( z.compareTo(max) > 0 )
      max = z; // z is the largest

    return max; // returns the largest object
}
```

Breadth-first search

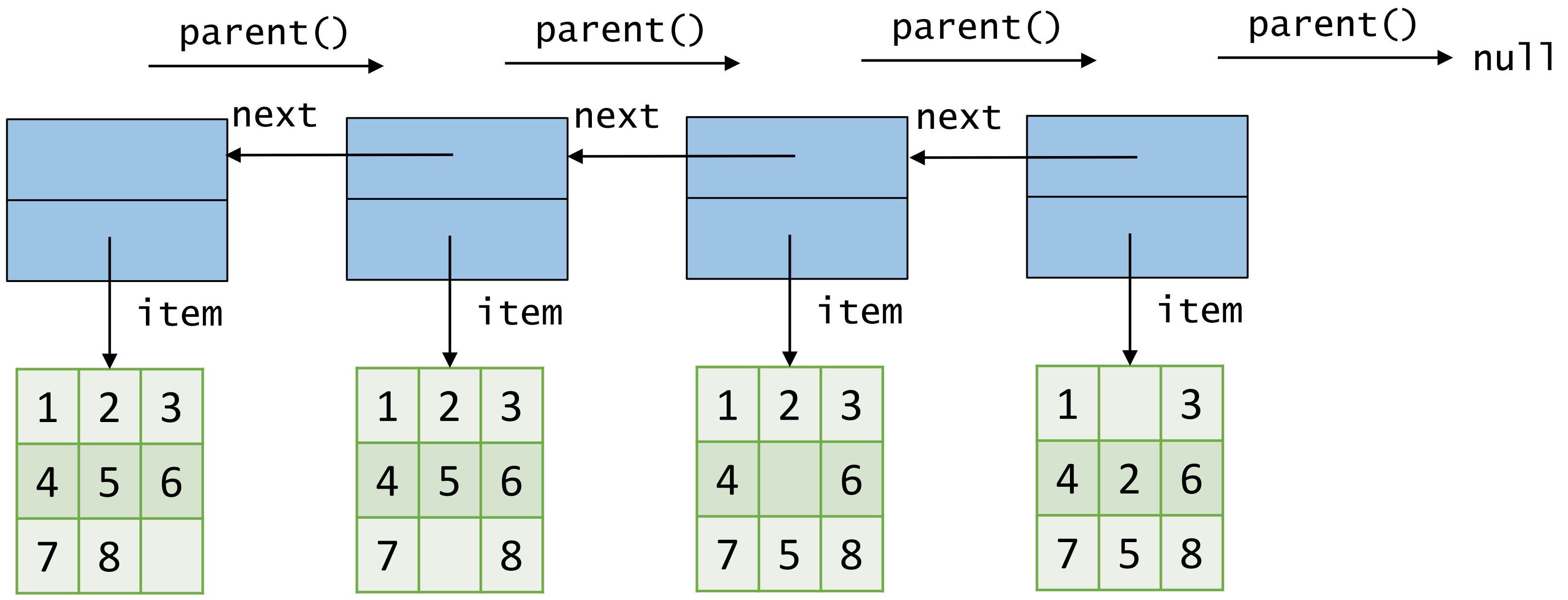
```
public class Solver {
 // Queue for maintaining configurations that are not visited yet.
    Queue<Configuration> toExamine;

    public String solve() {
        while ( ! toExamine.isEmpty() ) {
            Configuration next = toExamine.remove();
            if ( next.isSolution() ) {
                return "Success!";
            } else {
                for ( Configuration succ: next.successors() ) {
                    toExamine.add ( succ );
                }
            }
        }
        return "Failure!";
    }
}
```

A concrete puzzle

```
public class SlidingGame implements Configuration {
    public static final int N = 3, SIZE = N * N, HOLE = SIZE;

    private int [][] board;
    private int holeX, holeY;

    public SlidingGame( int [] start ) {
        board = new int[N][N];
        assert start.length == N*N : "Incorrect length";

        for( int p = 0; p < start.length; p++ ) {
            board[p % N][p / N] = start[p];
            if ( start[p] == HOLE ) {
                holeX = p % N;
                holeY = p / N;
            }
        }
    }
}
```

Showing a solution

Building a list structure (default method in `Configuration`)



(Hash) Sets

- Set: Collection with no duplicates
- `Set` is an interface
  - implementations: e.g., `HashSet`, `TreeSet`
- `HashSet` stores elements in a hash table

Membership testing in `HashSets`

- When testing whether a `HashSet` contains a given object, Java does this:
  - Java computes the hash code for the given object
  - Java compares the given object, using `equals`, *only* with elements in the set that have the *same* hash code
- Hence, an object will be considered to be in the set only if *both*:
  - It has the same hash code as an element in the set, *and*
  - the equals comparison returns true
- To use a `HashSet` properly, you must have a good `public boolean equals(Object)` and a good `public int hashCode()` defined for the *elements* of the set

Hashcode

- In the assignment: $$\text{hashvalue} = \sum_{x=0}^{N-1} \sum_{y=0}^{N-1} \text{board}[x][y] \cdot 31^{y+x \cdot N}$$
- Hint: Don't use `Math.power(31,y+x*N)` to compute $31^{y+x \cdot N}$

```
@Override
public int hashCode() {
    int hash = 0;
    for ( int x = N-1; x >= 0; x-- ) {
        for ( int y = N-1; y >= 0; y-- ) {
            hash = 31 * hash + board[x][y];
        }
    }
    return hash;
}
```

Horner's method

- See also: ItJP[Liang] chapter 27

Finally