Classes Part 2

Forest Agostinelli
University of South Carolina

Outline

- Enum classes
- Static methods and properties
- Breadth-first search

Enum

- An enumeration (“enum”) is a special kind of Class that only contains constants
- Used when creating a type that only has a set number of potential values
- Good programming practice to create in a separate Java File (like classes)
- The constant values are separated using a comma (“,”) and values should be capitalized
- Declare an enum just like any other class
  - Does not require construction
- Access the defined values using the dot (“.”)

Defining an Enum

```
public enum <<identifier>>{
        <<Value00>>,
        <<Value01>>,
        …
}
```

Example

```
enum PetType {CAT, DOG, HAMSTER, HEDGEHOG,
ARMADILLO, TURKEY, OWL, ABOMINATION};
```

Enum

- An enumeration (“enum”) is a special kind of Class that only contains constants
- Used when creating a type that only has a set number of potential values
- Good programming practice to create in a separate Java File (like classes)
- The constant values are separated using a comma (“,”) and values should be capitalized
- Declare an enum just like any other class
  - Does not require construction
- Access the defined values using the dot (“.”)

Declaring and Using an Enum

```
//Delare Enum
<<enum identifier>> <<id>>;
//Using
<<id>> = <<enum identifier>>.<<Value>>;
```

Example

```
PetType type;
type = PetType.DOG;
```

Enum Example

```java
/*
 * Written by JJ Shepherd
 */
public enum PetType {
    CAT,
    DOG,
    HAMSTER,
    HEDGEHOG,
    ARMADILLO,
    TURKEY,
    OWL,
    UNKNOWN
}
```

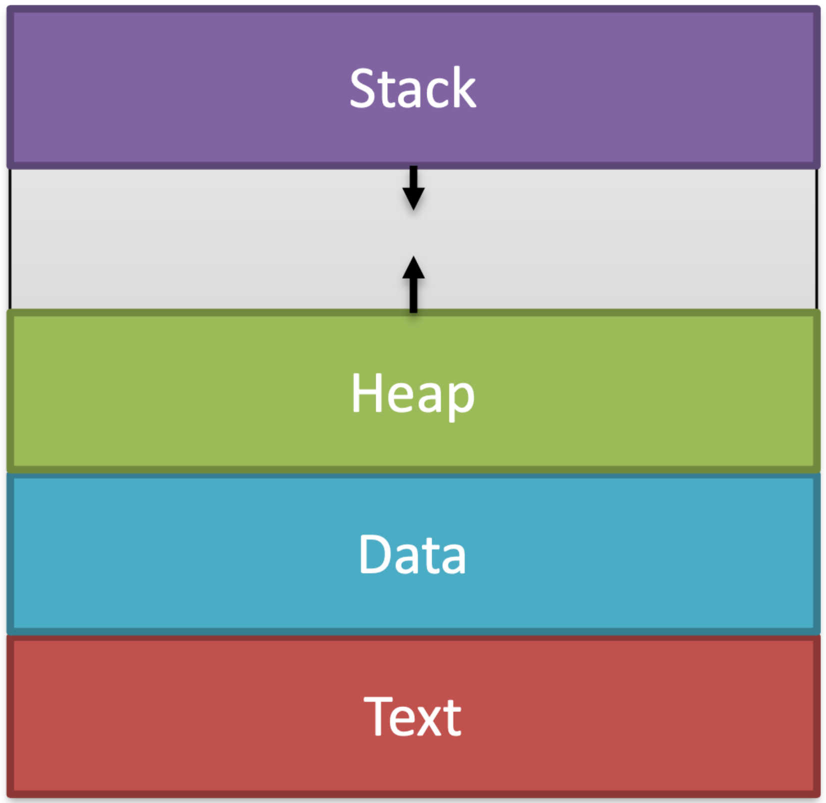
Outline

- Enum classes
- Static methods and properties
- Breadth-first search

Memory Allocation

- Programs have different sections of memory
  - Stack / Call Stack
  - Heap
  - Data (Global)
  - Text
- Methods are pushed on and popped off of the Stack
- Objects are Dynamically Allocated in the Heap
- The Stack and the Heap grow toward each other

<u>Process in Memory</u>



Static Properties

- Static methods and properties are created *statically*
  - Opposed to created *dynamically*
  - Created one time in the Data (Global) part of memory
- Static methods and properties are *shared* across all instances
  - Unlike dynamic methods or properties (instance variables) that are unique to each instance
- Uses the reserved word "static"
- CANNOT use the reserved word "this" to call static methods or properties
  - It only refers to dynamic instances

Static Properties

```
//Inside of a class
public static <<type>> <<id>>;
```

Example

```
public static int sharedInt;
```

Static Methods

- Static methods do not require an instance (object) to be called
  - Can be called directly from the Class
- Sometimes referred to as “Class Methods”
- Generally the scope is “public”
- Great to use when an *action* does not pertain to a particular instance (object)
  - Saves memory as it does not have to redefine the method for every instance. Only defined once.
- CANNOT use the reserved word “this” to call static methods or properties
  - It only refers to dynamic instances

Static Methods

```
public static <<return type>> <<id>> (<<parameters>>)
{
	//Body of the method
}
```

Example

```
//Assume inside the class “SimpleMath”
public static int addition(int a, int b)
{
	return a+b;
}
```

Static Methods

- Static methods do not require an instance (object) to be called
  - Can be called directly from the Class
- Sometimes referred to as “Class Methods”
- Generally the scope is “public”
- Great to use when an *action* does not pertain to a particular instance (object)
  - Saves memory as it does not have to redefine the method for every instance. Only defined once.
- CANNOT use the reserved word “this” to call static methods or properties
  - It only refers to dynamic instances

Calling Static Methods

```
<<Class Id>>.<<static method>>(<<parameters>>);
```

Example

```
int sum = SimpleMath.addition(2,3);
```

Static Methods

- Static methods can call other static methods
- Dynamic methods can call static methods
- Static methods CANNOT call dynamic methods directly
  - These methods can only be called when an instance (object) has been constructed
  - Just like for the Main Method
- Static methods can be called directly from the Main Method

Calling Static Methods

```
<<Class Id>>.<<static method>>(<<parameters>>);
```

Example

```
int sum = SimpleMath.addition(2,3);
```

Static Methods

- Commonly used Classes with Static Methods
  - Math
  - Wrapper Classes
- The class “Math” is built in to Java and provides many mathematic functions
  - Does not require an instance of Math to use methods
- Wrapper Classes like Integer, Double, Character
  - Provides common functionality and constants for primitive types
  - Very common is “.parseInt” or “.parseDouble”

Math Class Methods

Method	Return Type	Description	Example
pow(<<double>>,<< double>>)	Double	Power	Math.pow(2.0,3.0);
abs(<<A.N.T.>>)	A.N.T	Absolute Value	Math.abs(-7); Math.abs(-3.0);
max(<<A.N.T.>>, <<A.N.T>>)	A.N.T	Maximum Value between two values	Math.max(2,3); Math.max(3.5,2.5);
min(<<A.N.T.>>, <<A.N.T>>)	A.N.T	Minimum Value between two values	Math.max(2,3); Math.max(3.5,2.5);

A.N.T. = Any numeric type, such as int, double, float, or long

Static Methods

- Commonly used Classes with Static Methods
  - Math
  - Wrapper Classes
- The class "Math" is built in to Java and provides many mathematic functions
  - Does not require an instance of Math to use methods
- Wrapper Classes like Integer, Double, Character
  - Provides common functionality and constants for primitive types
  - Very common is ".parseInt" or ".parseDouble"

Math Class Methods

Method	Return Type	Description	Example
ceil(<<double>>)	Double	Ceiling (rounds up)	Math.ceil(2.1);
floor(<<double>>)	Double	Floor (rounds down)	Math.floor(3.9);
sqrt(<<double>>)	Double	Square root	Math.sqrt(4.0);
round(<<float>>)	Integer	Rounds up or down	Math.round(4.0f);
round(<<double>>)	Long	Rounds up or down	Math.round(4.0);

A.N.T. = Any numeric type, such as int, double, float, or long

Static Methods

- Commonly used Classes with Static Methods
  - Math
  - Wrapper Classes
- The class "Math" is built in to Java and provides many mathematical functions
  - Does not require an instance of Math to use methods
- Wrapper Classes like Integer, Double, Character
  - Provides common functionality and constants for primitive types
  - Very common is ".parseInt" or ".parseDouble"

Integer Class Methods and Properties

Method/Property	Return Type	Description	Example
MAX_VALUE	Integer	Returns $2^{31}-1$	Integer.MAX_VALUE
MIN_VALUE	Integer	Returns -2^{31}	Integer.MIN_VALUE
parseInt(<<String>>)	Integer	Converts String to Integer	Integer.parseInt("32")

Static Methods

- Commonly used Classes with Static Methods
  - Math
  - Wrapper Classes
- The class "Math" is built in to Java and provides many mathematical functions
  - Does not require an instance of Math to use methods
- Wrapper Classes like Integer, Double, Character
  - Provides common functionality and constants for primitive types
  - Very common is ".parseInt" or ".parseDouble"

Double Class Methods and Properties

Method/Property	Return Type	Description	Example
MAX_VALUE	Double	Returns Max Double Value	Double.MAX_VALUE
MIN_VALUE	Double	Returns Min Double Value	Double.MIN_VALUE
parseDouble (<<String>>)	Double	Converts String to Integer	Double.parseDouble ("32.0")

Static Methods

- Commonly used Classes with Static Methods
  - Math
  - Wrapper Classes
- The class “Math” is built in to Java and provides many mathematic functions
  - Does not require an instance of Math to use methods
- Wrapper Classes like Integer, Double, Character
  - Provides common functionality and constants for primitive types
  - Very common is “.parseInt” or “.parseDouble”

Character Class Methods

Method/Property	Return Type	Description	Example
toUpperCase(<<char>>)	Character	Converts character to upper case	Character.toUpperCase (‘a’);
toLowerCase(<<char>>)	Character	Converts character to lower case	Character.toUpperCase (‘A’);
isUpperCase(<<char>>)	Boolean	Tests for uppercase	Character.isUpperCase(‘a’);
isLowerCase(<<char>>)	Boolean	Tests for lowercase	Character.isLowerCase(‘a’);

Static Methods

- Commonly used Classes with Static Methods
  - Math
  - Wrapper Classes
- The class “Math” is built in to Java and provides many mathematical functions
  - Does not require an instance of Math to use methods
- Wrapper Classes like Integer, Double, Character
  - Provides common functionality and constants for primitive types
  - Very common is “.parseInt” or “.parseDouble”

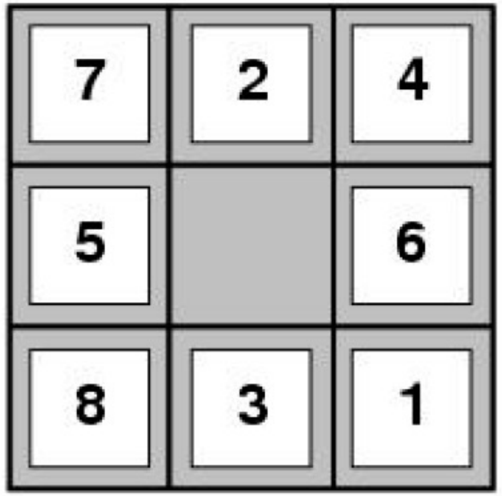
Character Class Methods

Method/Property	Return Type	Description	Example
isLetter(<<char>>)	Boolean	Tests for letter	Character.isLetter(‘a’);
isDigit(<<char>>)	Boolean	Tests for digit	Character.isDigit(‘a’);
isWhitespace(<<char>>)	Boolean	Tests for space such as ‘ ’, ‘\t’, and ‘\n’	Character.isWhitespace(‘ ’);

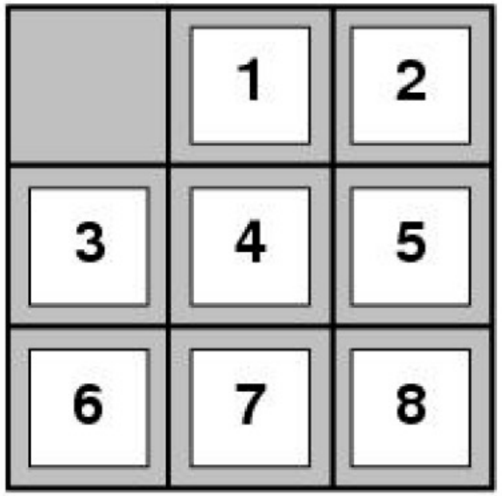
Outline

- Enum classes
- Static methods and properties
- Breadth-first search

Case Study: 8-puzzle



Start state



Goal state

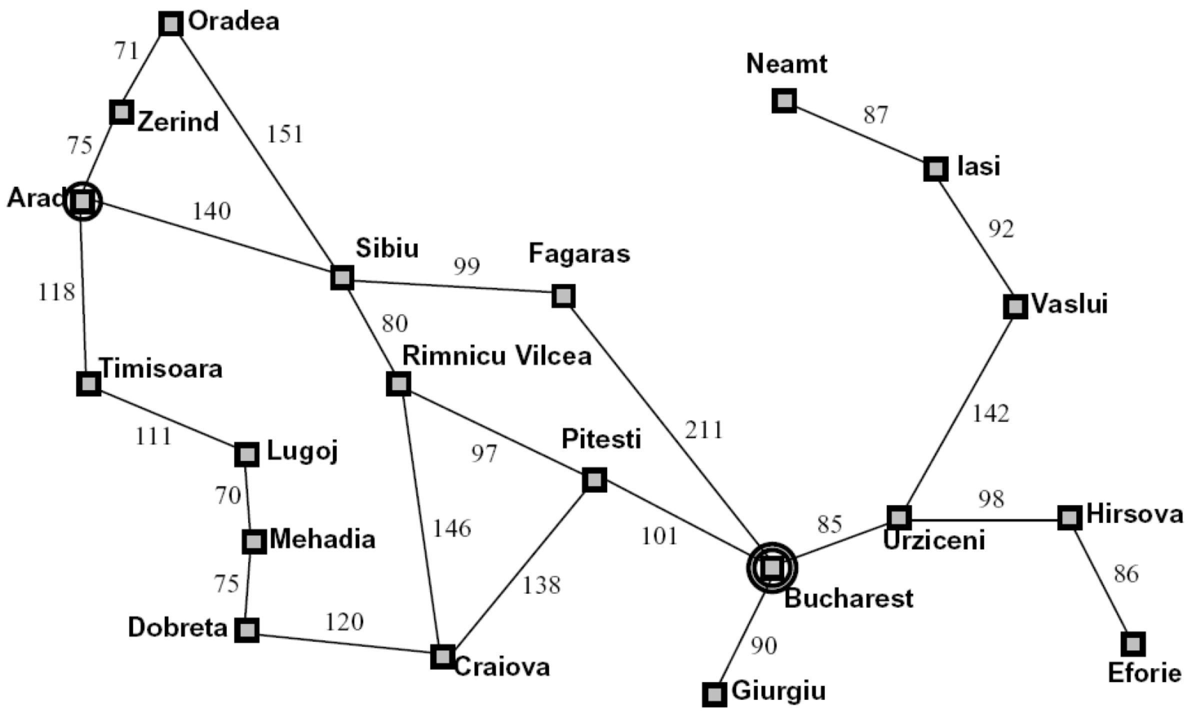
- Actions: swap the empty position with any tile that is horizontally or vertically adjacent
- 1.8×10^5 possible states (configurations)
- Larger versions
  - 15-puzzle: 1.0×10^{13} states
  - 24-puzzle: 7.7×10^{24} states
  - 35-puzzle: 1.8×10^{41} states
  - 48-puzzle: 3.0×10^{62} states

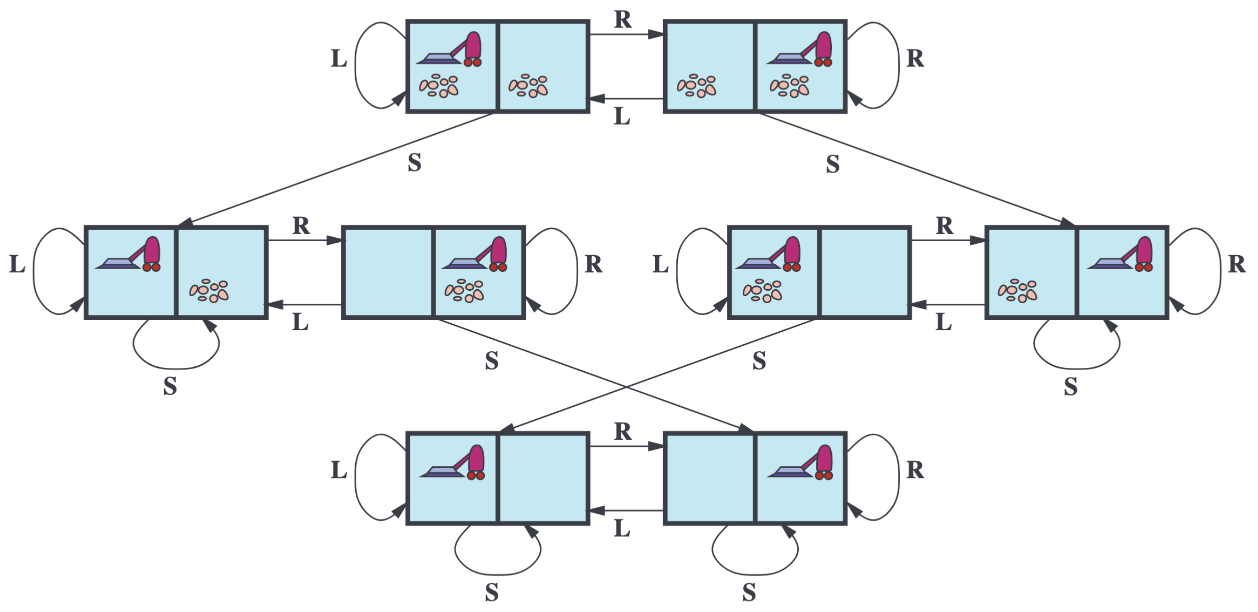
Defining a Classical Planning Problem

- **States $\mathcal{S}$**
  - Only keeps the details needed to solve the problem
- **Actions $\mathcal{A}$**
  - It is not always the case that every action can be taken in every state
- **Start state s_0**
- **Goal states $\mathcal{G} \subseteq \mathcal{S}$**
- **Transition model**
  - $s' = A(s, a)$
- **Transition cost function $c(s, a, s')$**
- Find a path from state s_0 to a state $s_g \in \mathcal{G}$
  - A minimum cost path is also referred to as an optimal or shortest path
  - There can be more than one optimal path

State Space Graph

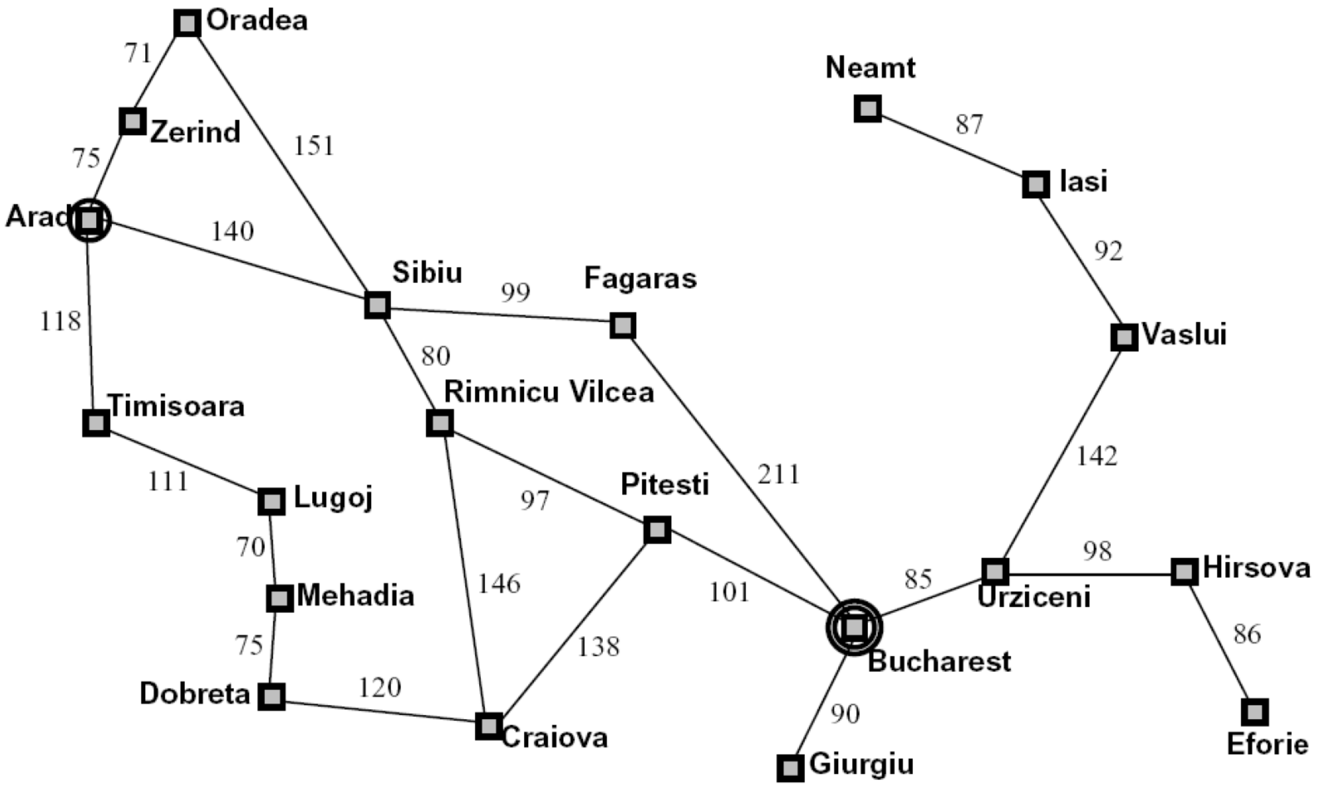
- Vertices: States
- Directed Edges: Actions
- Each state appears only once
- Pathfinding algorithms can be seen as finding a path between nodes in a graph





Example: Traveling in Romania

- Travel from Arad to Bucharest
- **States**
  - Cities
- **Actions**
  - Go to an adjacent city
- **Start state**
  - Arad
- **Goal state(s)**
  - Bucharest
- **Transition model**
  - Go to selected city
- **Transition cost function**
  - Driving time



Search Algorithms

- Expand nodes according to some priority until a goal node is selected for expansion
- Use a priority queue to sort nodes according to priority
  - This is referred to as OPEN or the “fringe”
  - For some algorithms, it can be implemented as a simple FIFO or LIFO queue
- Some algorithms use a CLOSED set to remember the nodes that have been generated
  - Sometimes referred to as “reached”
  - Prevents redundant node expansions

Nodes

- **Node:** Bookkeeping data structure for search
  - State
  - Parent node
  - Action
    - Action that the parent took to generate this node
  - Path cost
    - Cost of path from the start node to current node
- There can be multiple nodes with the same state
- We will refer to a node with the start state as n_0 and with a goal state as n_g
- A node is expanded when we use the transition function to generate all its children

Node Expansion

- Apply every possible action to the state associated with the node

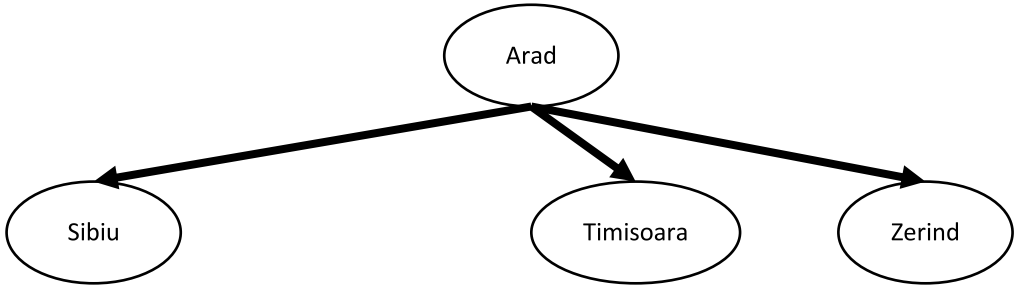
for each action a for $n.s$

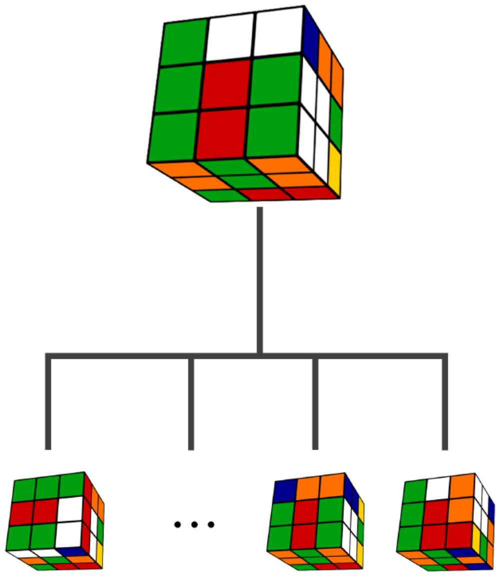
$s' = A(n.s, a)$ // next state

$g = n.g + c(n.s, a, s')$ // path cost

$d = n.d + 1$ //depth

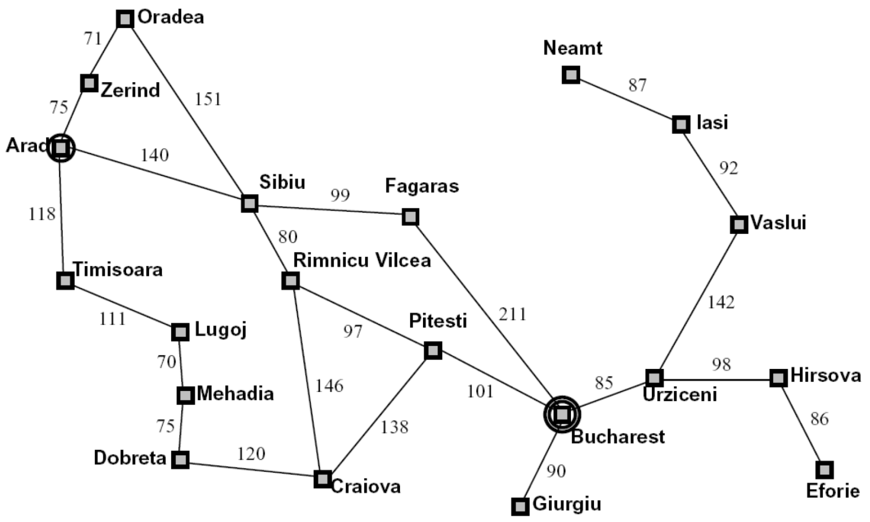
$n_c = Node(s', n, a, g, d)$ //new node



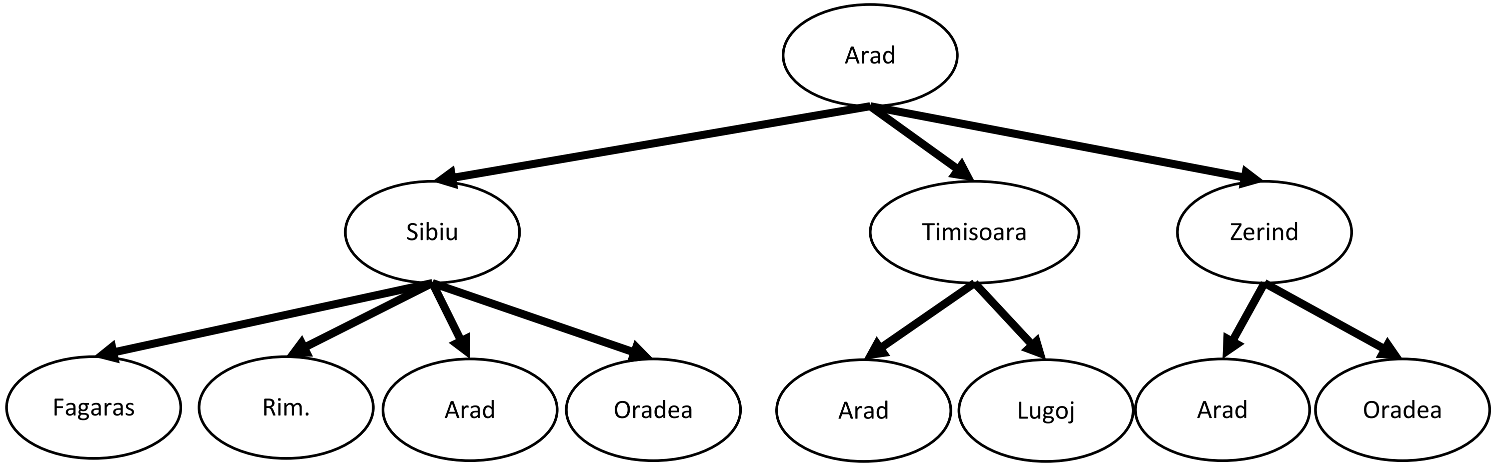


Search Tree

- Pathfinding algorithms can form a tree where states appear multiple times; representing different paths one can take to the same state
  - Remember, every node except for the root node has exactly one parent
- Vertices: States
- Directed Edges: Actions



State space graph



Search tree

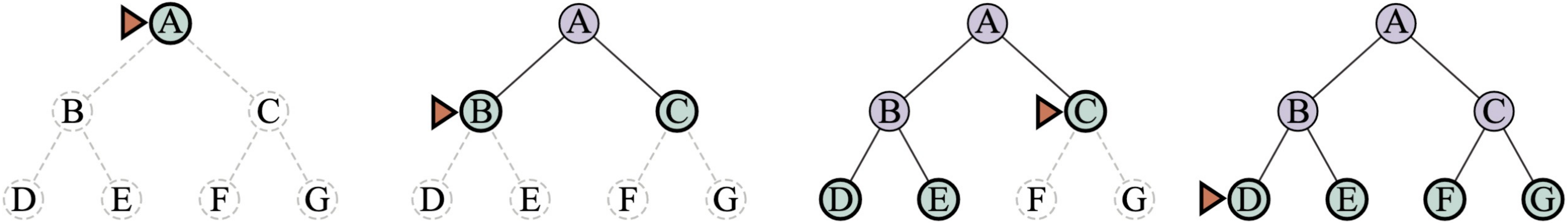
Breadth First Search

function BREADTH-FIRST-SEARCH(*problem*) **returns** a solution node or *failure*
  node ← NODE(*problem*.INITIAL)
  if *problem*.IS-GOAL(*node*.STATE) **then return** *node*
  frontier ← a FIFO queue, with *node* as an element
  reached ← {*problem*.INITIAL}
  while not IS-EMPTY(*frontier*) **do**
    node ← POP(*frontier*)
    for each *child* **in** EXPAND(*problem*, *node*) **do**
      s ← *child*.STATE
      if *problem*.IS-GOAL(*s*) **then return** *child*
      if *s* is not in *reached* **then**
        add *s* to *reached*
        add *child* to *frontier*
  return *failure*

Breadth-first search is a special case where we can do the goal test when nodes are generated instead of when they are selected for expansion

Breadth-First Search

- Prioritize the shallowest nodes
- For breadth-first search, we do not have to wait until the goal node is selected for expansion, we can terminate when the goal state is generated



Node to be expanded next

Not yet generated

In OPEN

Expanded