Advanced Software Development 1

Dr. Eliahu Khalastchi

2017

המסלול האקדמי
המכללה למינהל

From Pseudo code to Object Oriented Java code

Let's learn how to code an algorithm with OO design in mind…

We will learn how to decouple an algorithm form the problem it solves
Using the "dependency injection" technique

Let’s say we want to solve these problems

- We refer to each problem setting as a domain

Unscramble Domain:

LI oJe vvaa

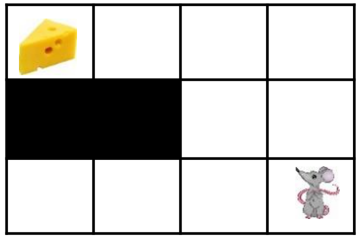
I Love Java

- A state represents the current string
- The cost to switch two letters can be derived from their distance
- Which switches do we need?

Parking Lot Domain:

- A state represents current car positions
- Each car may have a different “move” value
- How can we get the black car out?

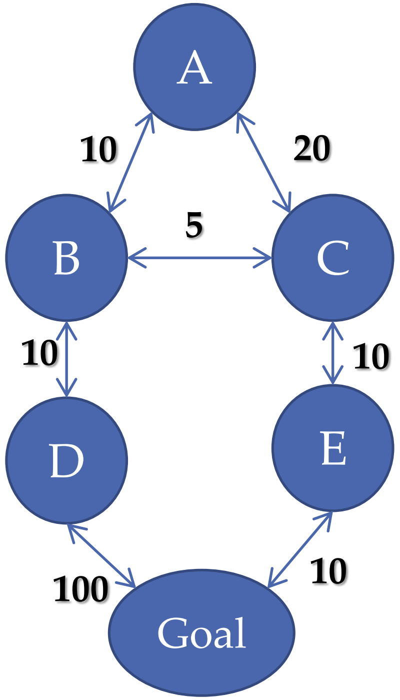
Maze Domain:



- A state may represent the position of the mouse
- The cost to move diagonally can be 15
- The cost to move directly can be 10
- What is the cheapest path?

We can use the **Best First Search** algorithm

Best First Search:
OPEN = [initial state]   // a **priority queue** of states to be evaluated
CLOSED = []   // a **set** of states already evaluated
while OPEN is not empty
do
  1. **n** ← dequeue(OPEN) // Remove the best node from OPEN
  2. add(n,CLOSED)   // so we won't check n again
  3. If **n** is the goal state,
      backtrace path to **n** (through recorded parents) and return path.
  4. Create **n**'s successors.
  5. For each successor **s** do:
    a. If **s** is not in CLOSED and **s** is not in OPEN:
      i. update that we came to **s** from **n**
      ii. add(**s**,OPEN)
    b. Otherwise, if this new path is better than previous one
      i. If it is not in OPEN add it to OPEN.
      ii. Otherwise, adjust its priority in OPEN
done



OPEN

CLOSED

A {0,null}
B {10,A}
~~C {25,A}~~ C {15,B}
D {20,B}
E {25,C}
G {120,D} G {35,E}

Goal ← E ← C ← B ← A

How can we do it in the OOP way?

We don’t want to implement the algorithm for a specific domain because would have to duplicate the code for every domain!!

We **do not** just implement it inside one class…

- We need to think about the **design!!!**
- We need to think how this algorithm is going to be used
  - In different projects
  - For different problems

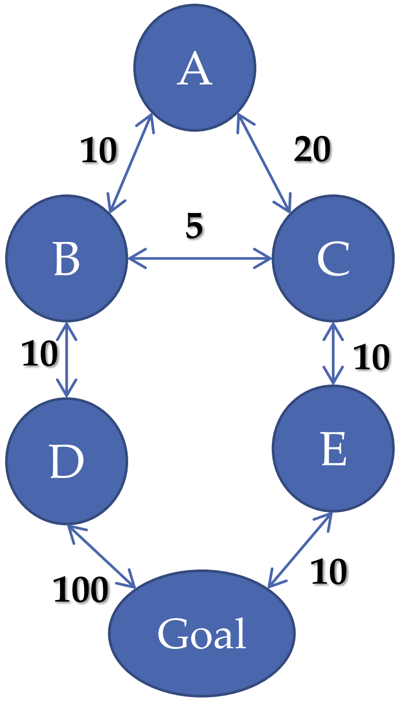
- Where to start?
- It is probably a good idea to suppurate items which are
  - **Domain Specific** - specific to a certain domain e.g., the cost to move the mouse
  - **Domain Independent** - not specific to a certain domain
- We want to create a **domain independent** implementation of the BFS algorithm

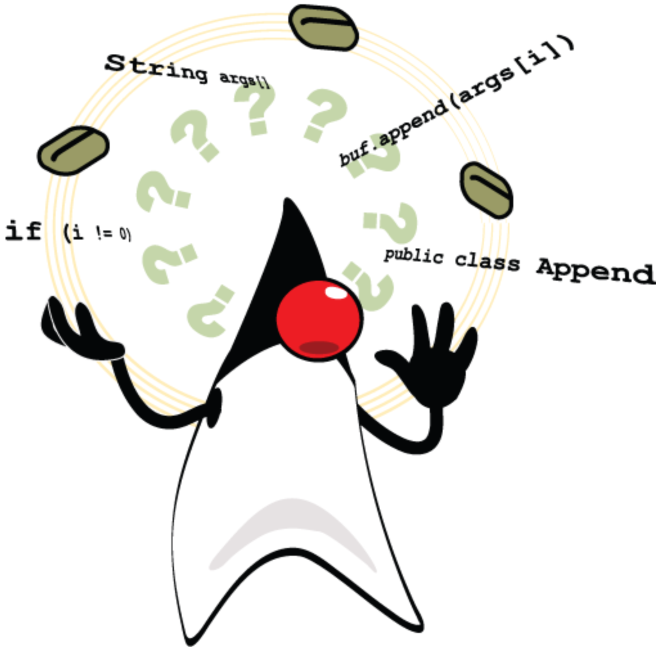
Decouple the algorithm from the problem it solves!

- We need to start by defining what we expect from
  - A general search problem
  - A general search algorithm
- This expectation is the functionality of these entities
- In other words, we should define their interfaces
- Later, each entity can have different independent implementations

Defining the Problem

The search domain





We can start with a general **State** class

Since each node represents a “state” of the problem, we can create a State class
Inside, we can use whatever types that can describe states generally

```
public class State {
    private String state;    // the state represented by a string
    private double cost;     // cost to reach this state
    private State cameFrom;  // the state we came from to this state

    public State(String state){    // CTOR
        this.state = state;
    }

    @Override
    public boolean equals(Object obj){ // we override Object's equals method
        return state.equals(((State)obj).state);
    }
   // ...
}
```

We can start with a general **State** class

Since each node represents a “state” of the problem, we can create a State class
Inside, we can use whatever types that can describe states generally

```
public class State {
    private String state;    // the state represented by a string
    private double cost;     // cost to reach this state
    private State cameFrom;  // the state we came from to this state

    public State(String state){    // CTOR
        this.state = state;
    }


    public boolean equals(State s){ // it’s easier to simply overload
        return state.equals(s.state);
    }
   // ...
}
```

Example of Use:

```
State a, b, goal;
a = new State("A");
b = new State("B");
goal=new State("B");

System.out.println(b.equals(goal));
// true
```

Now, let's focus on the cost issue…

- Who should calculate the cost to move from one state to another??
  - Should it be a method of State? ✖
  - Should it be a static method of State? ✖

- No! the cost calculation is **domain specific**!
  - i.e., relevant to a specific problem and not a general solution

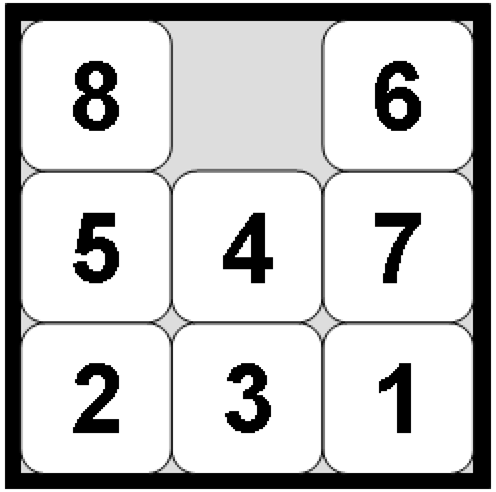
- It should not be implemented in the State class
- But rather the cost calculation in a domain-specific class

Specific states can be represented by the String variable

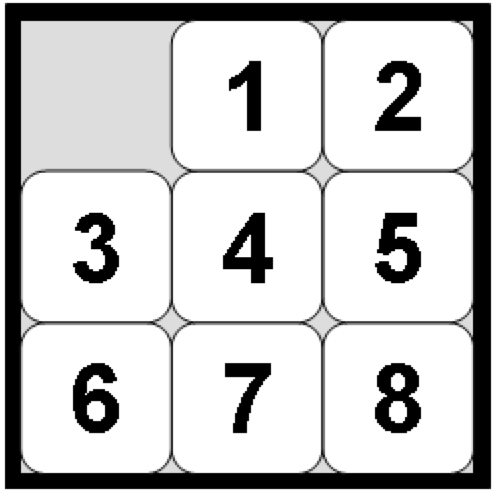
Domain independent

State
String state

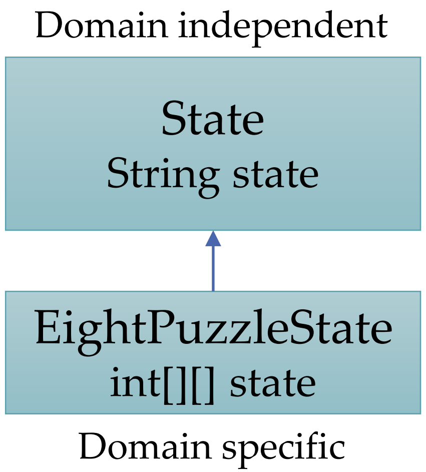
State s1=new State();
s1.setState("8 6547231");



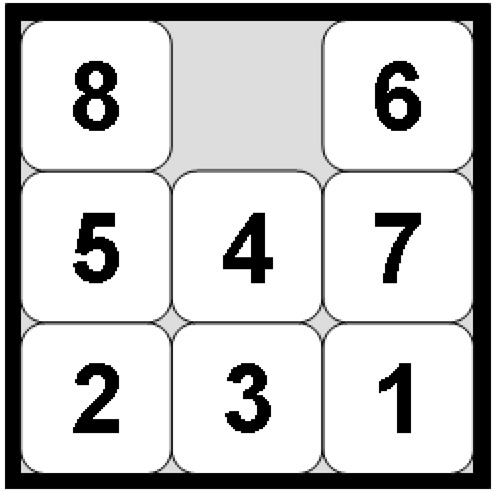
" 12345678"



Specific states can be overloaded in an extended class




```
EightPuzzleState s1=new EightPuzzleState();
int[][] state={{8,0,6},{5,4,7},{2,3,1}}”;
s1.setState(state);
```



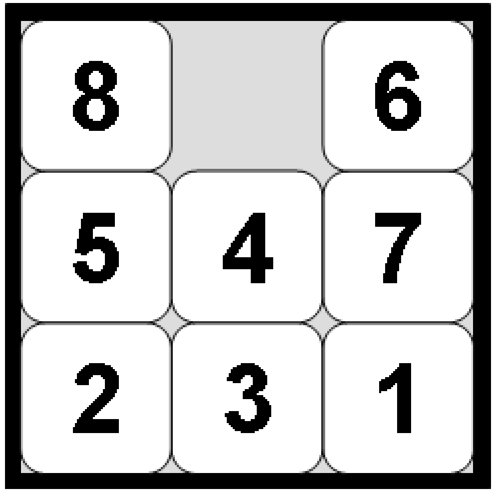
Specific states can be instantiated from a generic State class

Domain independent

State<T>

```
public class State<T> {
 T state;
 public State(T state) {
  this.state=state;
 }
...
}
```

State<String> s1=new State<String>(“8 6547231”);



What do we expect from a search problem?

Best First Search:

OPEN = [initial state]   // a **priority queue** of states to be evaluated
CLOSED = []   // a **set** of states already evaluated
while OPEN is not empty
do

1. **n** ← dequeue(OPEN) // Remove the best node from OPEN
2. add(n,CLOSED)   // so we won't check n again
3. If **n** is the goal state,
   backtrace path to **n** (through recorded parents) and return path.
4. Create **n**'s successors.
5. For each successor **s** do:
   a. If **s** is not in CLOSED and **s** is not in OPEN:
      i. update that we came to **s** from **n**
      ii. add(**s**,OPEN)
   b. Otherwise, if this new path is better than previous one
      i. If it is not in OPEN add it to OPEN.
      ii. Otherwise, adjust its priority in OPEN

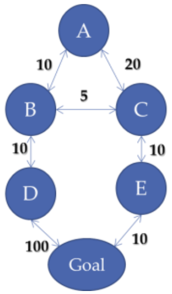
done

What do we need from the search domain?

- To give us the ***start state*** and the ***goal state***
- Given a state, what are the possible states we can get to?
  - This is actually a part of the graph…

```
public interface Searchable {
   State getInitialState();
   State getGoalState();
   ArrayList<State> getAllPossibleStates(State s);
}
```

State

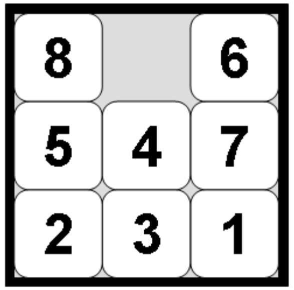


EightPuzzle

EightPuzzleState

getAllPossibleStates of

“8 6547231”



Returns:

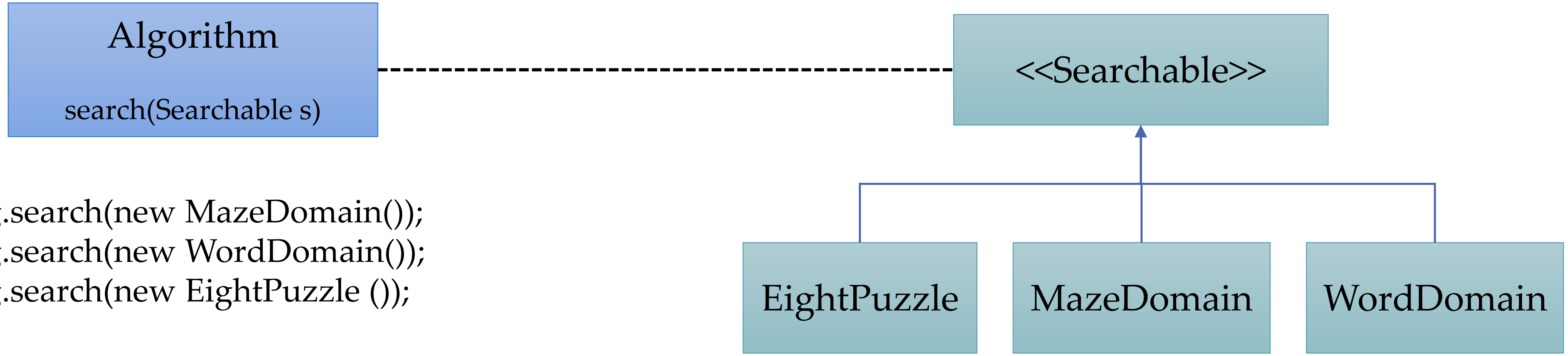
“ 86547231”

“8465 7231”

“86 547231”

What did we gain?

- The algorithm just knows Searchable (and State)
- It does not know any specific searchable domains (and states)
- ➔ we can switch searchable domains independently from the algorithm!!!




```
alg.search(new MazeDomain());
alg.search(new WordDomain());
alg.search(new EightPuzzle ());
```

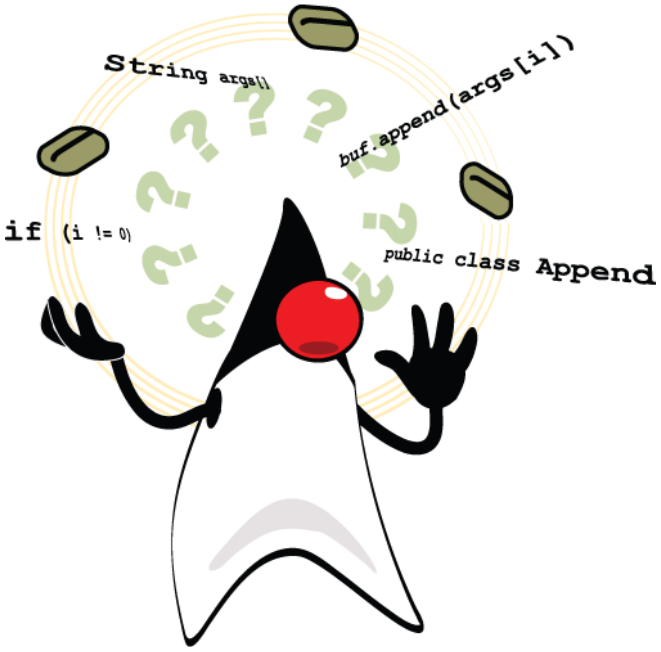
We call this design ***dependency injection***

Defining the Algorithm

The search algorithm

Best First Search:
OPEN = [initial state] // a **priority queue** of states to be evaluated
CLOSED = [] // a **set** of states already evaluated
while OPEN is not empty
do
1. **n** ← dequeue(OPEN) // Remove the best node from OPEN
2. add(n,CLOSED) // so we won't check n again
3. If **n** is the goal state,
   backtrace path to **n** (through recorded parents) and return path.
4. Create **n**'s successors.
5. For each successor **s** do:
   a. If **s** is not in CLOSED and **s** is not in OPEN:
      i. update that we came to **s** from **n**
      ii. add(**s**,OPEN)
   b. Otherwise, if this new path is better than previous one
      i. If it is not in OPEN add it to OPEN.
      ii. Otherwise, adjust its priority in OPEN

done



By now, we are able to switch searchable domains independently from the searching algorithm.

We want to be able to switch searching algorithms as well.

What more do we need to think about?

- The BFS is a **type of** “searcher”
- We may want to implement **other** searching algorithms in the **future** as well
  - E.g., beam search, A*, hill climbing, etc.
- We want our system to work with any type of “searcher”
  - I.e., we can replace the searching algorithm without changing the system’s code
- For that we need to define an Interface!

```
public interface Searcher {
    // the search method
    public Solution search(Searchable s);
    // get how many nodes were evaluated by the algorithm
    public int getNumberOfNodesEvaluated();
}
```

The expected common functionality to any searcher

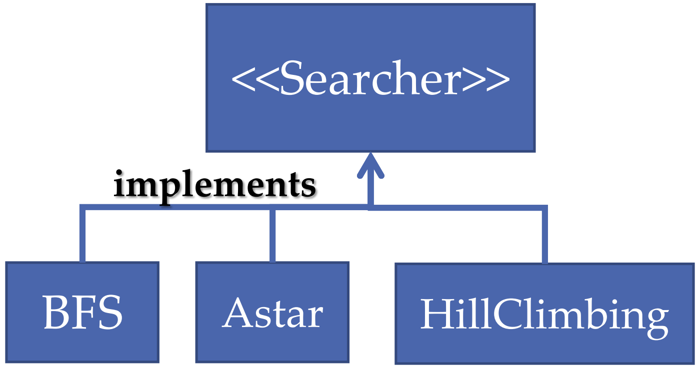
The client only needs to know the interface

- The “client” is another programmer, that may use the searching algorithms

```
public void testSearcher(Searcher searcher, Searchable searchable){

   Solution sol=searcher.search(searchable);
   int n = searcher.getNumberOfNodesEvaluated();
   // ...
}
```

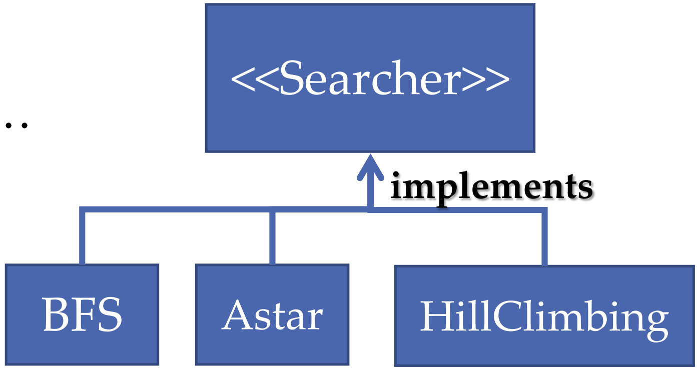
Dependency Injection!



Examples:
tester.testSearcher(new BFS(), new EightPuzzle());
tester.testSreacher(new Astar(), new EightPuzzle());
tester.testSearcher(new HillClimbing(), new MazeDomain());

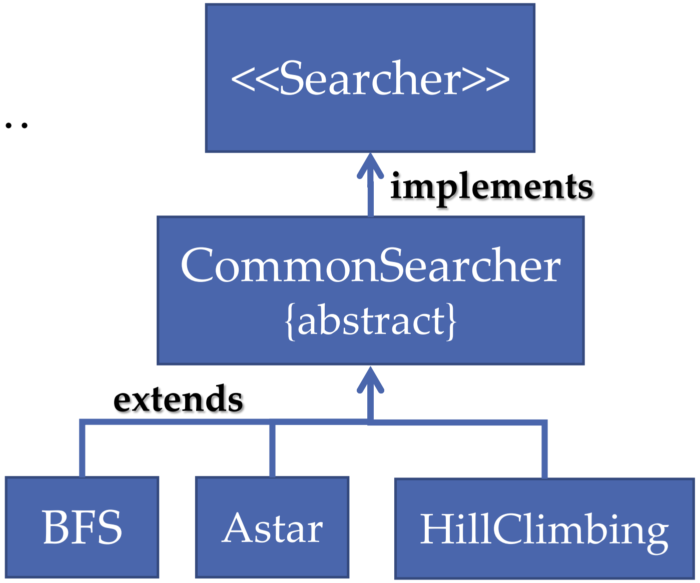
What all searchers have in common?

- Let’s assume that all searching algorithms have a priority queue of states
- It will be wasteful to implement it over and over again with each algorithm…
- We can use an abstract class!



What all searchers have in common?

- Let’s assume that all searching algorithms have a priority queue of states
- It will be wasteful to implement it over and over again with each algorithm…
- We can use an abstract class!




```
public abstract class CommonSearcher implements Searcher {

 protected PriorityQueue<State> openList;
 private int evaluatedNodes;

 public CommonSearcher() {
  openList=new PriorityQueue<State>();
  evaluatedNodes=0;
 }

 protected State popOpenList() {
  evaluatedNodes++;
  return openList.poll();
 }
```

```
 @Override
 public abstract Solution search(Searchable s);

 @Override
 public int getNumberOfNodesEvaluated() {
  return evaluatedNodes;
 }
}
```

BFS – best first search

```
Best First Search:
OPEN = [initial state]   // a priority queue of states to be evaluated
CLOSED = []              // a set of states already evaluated
while OPEN is not empty
do
  1. n ← dequeue(OPEN) // Remove the best node from OPEN
  2. add(n,CLOSED)       // so we won't check n again
  3. If n is the goal state,
       backtrace path to n (through recorded parents) and return path.
  4. Create n's successors.
  5. For each successor s do:
         a. If s is not in CLOSED and s is not in OPEN:
                i.  update that we came to s from n
                ii. add(s,OPEN)
         b. Otherwise, if this new path is better than previous one
                i.  If it is not in OPEN add it to OPEN.
                ii. Otherwise, adjust its priority in OPEN
done
```

The BFS search algorithm

```
public Solution search(Searchable s) {
  addToOpenList(s.getInitialState());
  HashSet<State> closedSet=new HashSet<State>();

  while(openList.size()>0){
    State n=popOpenList();// dequeue
    closedSet.add(n);

    if(n.equals(s.getGoalState()))
      return backTrace(s.getGoalState(), s.getStartState());
      // private method, back traces through the parents

    ArrayList<State> successors=s.getAllPossibleStates(n) //however it is implemented
    for(State state : successors){
      if(!closedSet.contains(state) && ! openListContains(state)){
        state.setCameFrom(n);
        addToOpenList(state);
      } else{
        //...
```

Our BFS implementation is domain independent! ☺

Summary

1. Think of the abstract problem representation
   1. e.g., shortest path in a graph
2. Find or create an algorithm that solves the abstract problem
   1. e.g., Best First Search
3. The required domain-specific information defines the functionality of problem type
   1. Put in an interface, e.g., Searchable
4. Define the solver's interface, e.g., Searcher
   1. Make sure to receive the abstract problem type via **dependency injection**
5. Define in **abstract classes** the common code for algorithm implementations
6. Extend these abstract classes to define a specific code for a specific algorithm