AI Theory, W3/L2

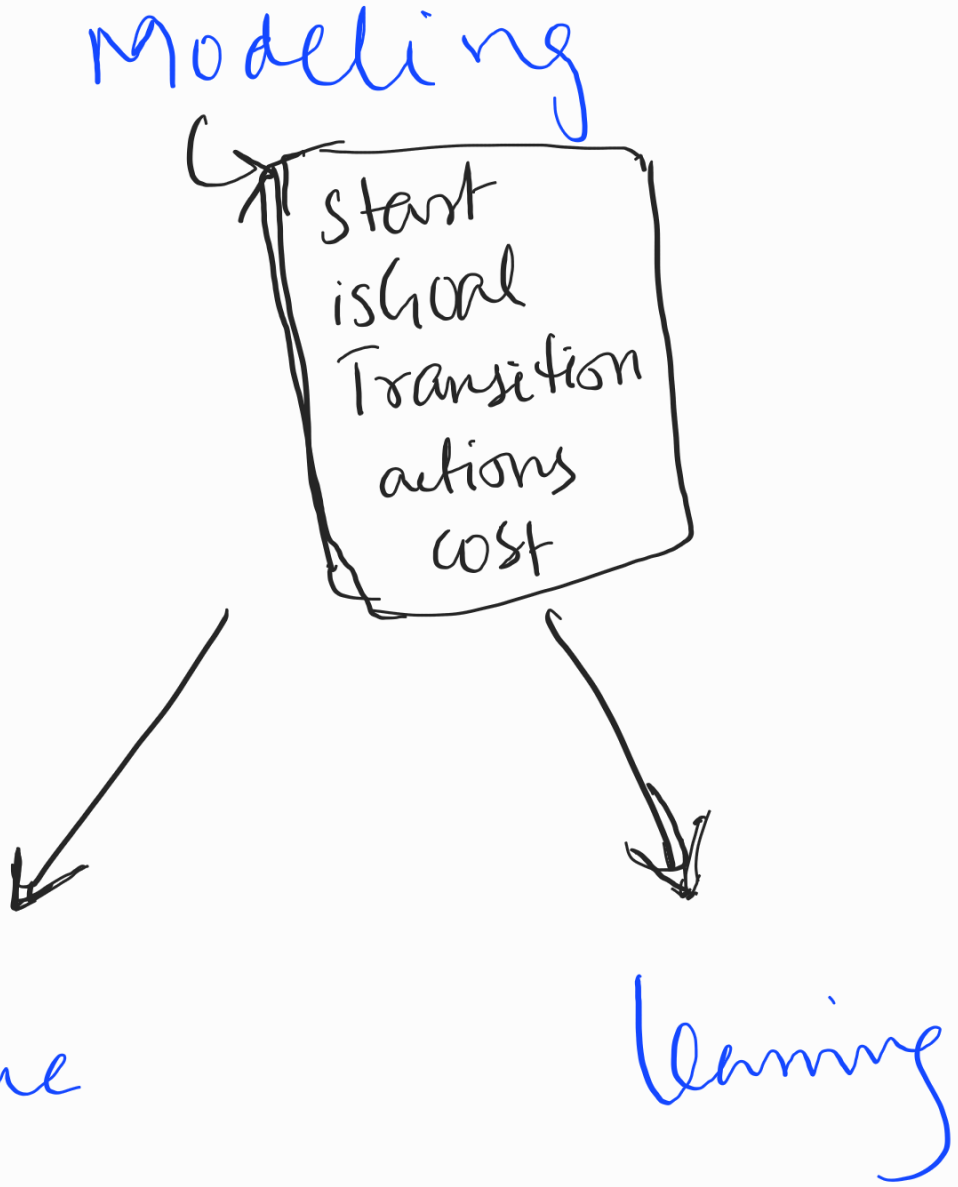
Recall backtrack search algo.

```
def Backtrack_Search(s):

    // Code
```

we will not pass the state to the function (this approach implies that Backtrack_Search is a attribute of the Transportation class we made earlier). We are not gonna do this.

⇒ Infact we will use problem object to this function.



our code (solution) should be generic. The functions in box are templates for all probs.

for $n=6$, cost $=4$

walk, walk, tram

cost $\rightarrow$ $1 + 1 + 2 = 4$

for $n=100$, cost $=13$

W W T T T W T T

$\rightarrow$ 1 1 2 2 2 1 2 2

for $n=200$, code in die down mode

why?

Tree branches grow exponentially at $n=200$

How do we apply Dynamic Programming?

```
def BackTrack_S(prob):
    min_cost, history = ∞, [ ]
    cache = { }
    def recurse(s, c, hist):
        if prob.isGoal(s):
            if c < min_cost:
                min_cost = c
                history = hist
            return
        if s in cache:
            return cache[s]
```

```
else:
    actions = prob.actions(s)
    for a in actions:
    // Recursive calls

    ans = recurse(------)
    cache[new_s] = ans

recurse(prob.start(), 0, [])
return (min, hist)
```

<u>Magic</u>: now it gives the answer for n=1000 easily.

Bottom-up Solution:

2 - 1 offset solution

works ONLY for this solution.

but it isnt generic.

Goal se start & end at start.

Depth - first Search

⇒ Backtrack + stop at first instance of goal.

Legend : $\underline{b}$ actions per state, max. depth is D actions

Space = O(D)

Time = O(bD)

Assume: Action costs Cost(s,a) = 0

How do we update it?

```
def BackTrack_DFS(prob):
    min, history, isfoundGoal = ∞, [], false
```

All the same code.

except:

```
    if problem.isGoal(s):
        if c < min:
            " "
            isfoundGoal = True
```

Like recursion with sentinel

```
    if NOT isfondGoal:
```

if s in cache:

// rest of the same code

Breadth-first Search:

idea: explore all nodes in order of increasing breadth.

legend: d is depth, w width is max.

$O(b^d)$ Time

$O(d)$ space

```
def Backtrack_BFS(prob):
    # challenge to
      implement itself
```

```
def Backtrack_Iterative_Deepening
    search:
```

idea is that we progressively deepen our search level by level.

BFS, but with height passed to itself as parameter