

5. Search 2 - A Star Algorithm



Key idea: state

A **state** is a summary of all the past actions sufficient to choose future actions **optimally**.

past actions (all cities) 1 3 4 6 5 3

Uniform cost search

Uniform cost search is quite simlair with Dijkatra ? Dijkstra is a special case of A* Search Algorithm, where $h = 0$ for all nodes.

Theory: Guerentee the shortest path: It is guerenteed that when u pop up a point S from frontier The cost/path is guerenteed to be the minimum cost/shortest path

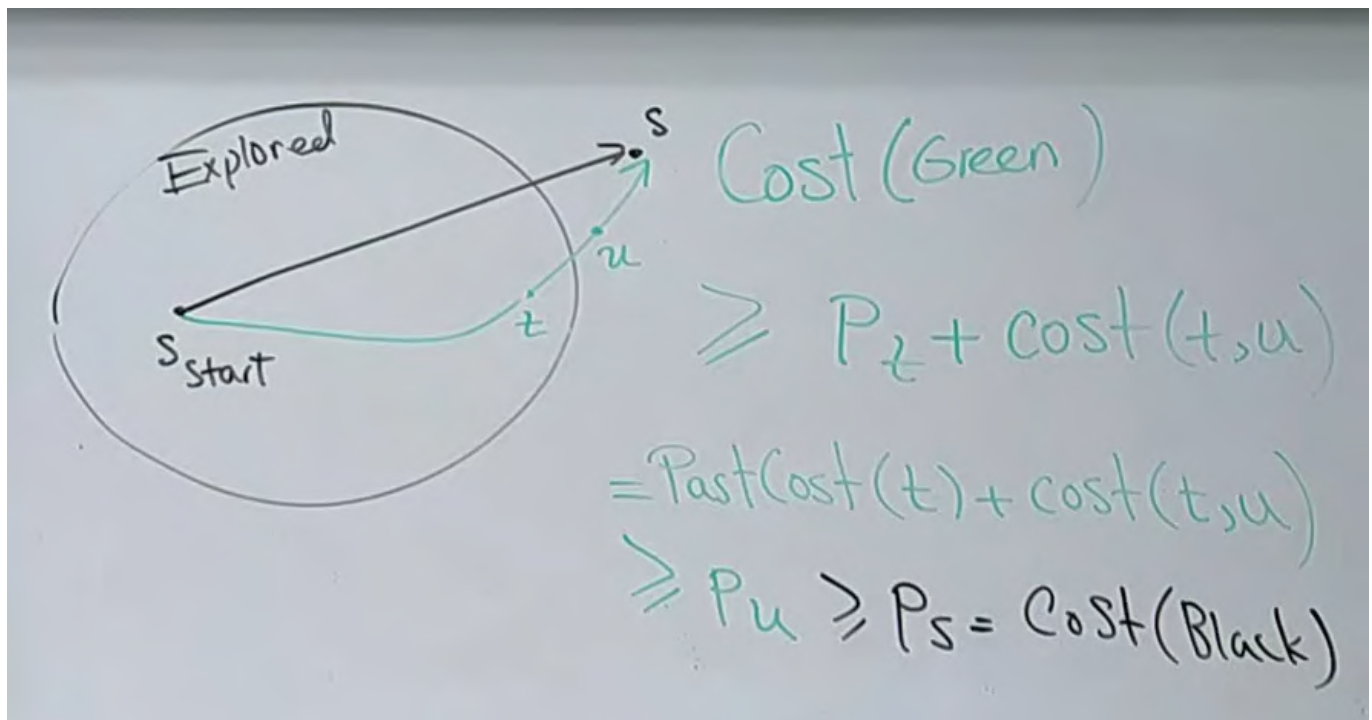
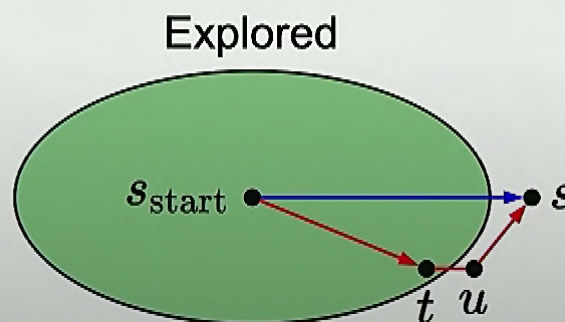
Analysis of uniform cost search



Theorem: correctness

When a state s is popped from the frontier and moved to explored, its priority is $\text{PastCost}(s)$, the minimum cost to s .

Proof:



$P_u \leq P_t + \text{Cost}(t, u)$ This is the characteristic of priority queue

DP versus UCS

N total states, n of which are closer than end state

Algorithm	Cycles?	Action costs	Time/space
DP	no	any	$O(N)$
UCS	yes	≥ 0	$O(n \log n)$

Note: UCS potentially explores fewer states, but requires more overhead to maintain the priority queue

Note: assume number of actions per state is constant (independent of n and N)

If u meet a graph with cycle and negative values, you can also use bellman ford algorithm (Beyond the scope of this class)

Learning in Search problems

Back to our tram problem

Search

Transportation example

Start state: 1

Walk action: from s to $s + 1$ (cost: 1)

Tram action: from s to $2s$ (cost: 2)

End state: n



search algorithm

walk walk tram tram tram walk tram tram
(minimum cost path)

However modeling is really hard, in many real world scenarios we don't know what the costs are

Learning costs

If we know the optimal path already So learning will predict what are the costs are, based on the optimal path



Learning

Transportation example

Start state: 1

Walk action: from s to $s + 1$ (cost: ?)

Tram action: from s to $2s$ (cost: ?)

End state: n

walk walk tram tram tram walk tram tram



learning algorithm

walk cost: 1, tram cost: 2

I wanna learn walk is 1 tram is 2

Another example

We wanna learn about ppl picking up the bottle We really don't know about the cost function However we can track the path that they took and learn the cost function from it.

Learning as an inverse problem

Forward problem (search):

$$\text{Cost}(s, a) \longrightarrow (a_1, \dots, a_k)$$

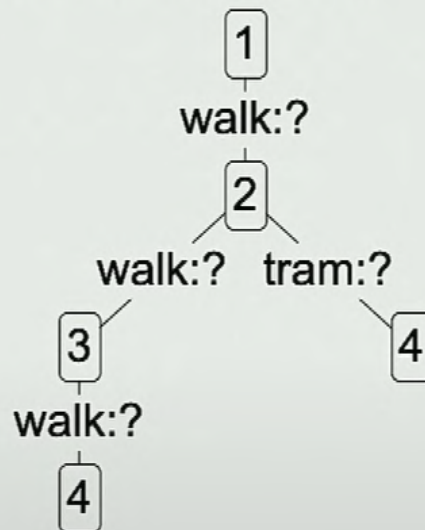
Inverse problem (learning):

$$(a_1, \dots, a_k) \longrightarrow \text{Cost}(s, a)$$

Searching: find the path with given cost functions Learning: Learn the cost function from given paths (inverse of search)

Prediction (inference) problem

Input x : search problem without costs

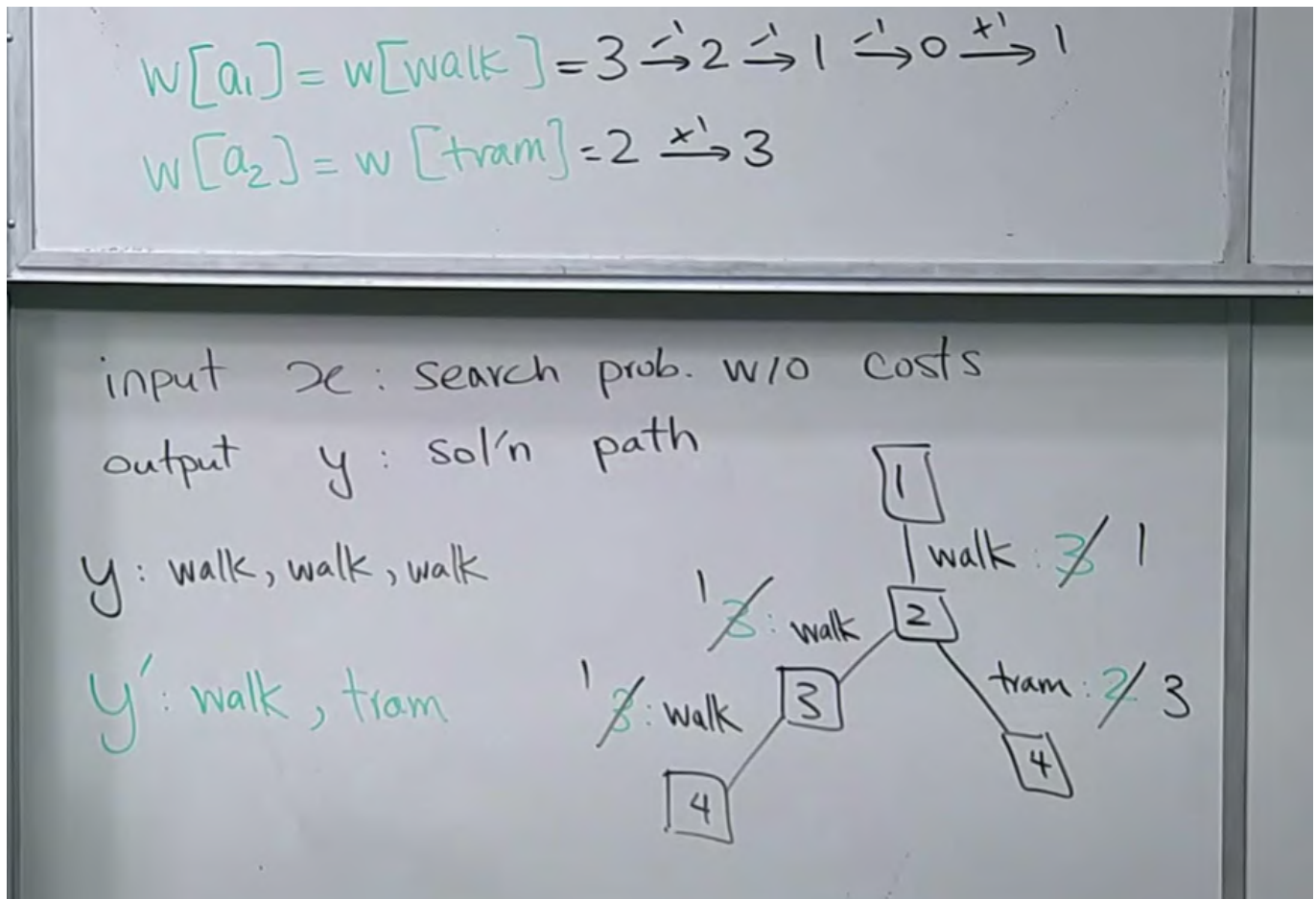


How do we learn it

Input X : The search problem without costs Output Y : solution path

$x \Rightarrow$ Predictor $F \Rightarrow Y$

A simplest example - Structured Perceptron



Let's say the weights only depends on the action (In real it can depend on state too)

We can randomly initialize the weights, and try prediction

After predicted y' , we'll check y and y' we'll decrease the weight (meaning decrease the cost) (by 1 for example) for what's in real y , because we want the cost of true thing to be small We'll increase the weight (meaning increase the cost) (by 1 for example) for what's in y'

We are doing this because we want to match the prediction

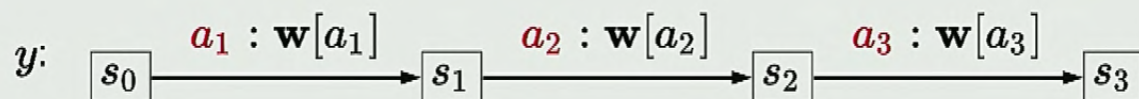
After predicted y' , we'll see how different is it form y If action are the same in y and y' , the increased weight = decreased weight, the weight won't change(CUZ it's already the ture answer) If we walk more in real y , the weight decreased will be more Therefore, after such operation, we'll get closer to the true weight

Modeling costs (simplified)

Assume costs depend only on the action:

$$\text{Cost}(s, a) = \mathbf{w}[a]$$

Candidate output path:



Path cost:

$$\text{Cost}(y) = \mathbf{w}[a_1] + \mathbf{w}[a_2] + \mathbf{w}[a_3]$$

w^*a is the cost

Learning algorithm



Algorithm: Structured Perceptron (simplified)

- For each action: $\mathbf{w}[a] \leftarrow 0$
- For each iteration $t = 1, \dots, T$:
 - For each training example $(x, y) \in \mathcal{D}_{\text{train}}$:
 - Compute the minimum cost path y' given \mathbf{w}
 - For each action $a \in y$: $\mathbf{w}[a] \leftarrow \mathbf{w}[a] - 1$
 - For each action $a \in y'$: $\mathbf{w}[a] \leftarrow \mathbf{w}[a] + 1$
- Try to decrease cost of true y (from training data)
- Try to increase cost of predicted y' (from search)


```

70 trueWeights = {'walk': 1, 'tram': 2}
71 return [(N, predict(N, trueWeights)) for N in range(1, 10)]
72
73 def structuredPerceptron(examples):
74     weights = {'walk': 0, 'tram': 0}
75     for t in range(100):
76         numMistakes = 0
77         for N, trueActions in examples:
78             # Make a prediction
79             predActions = predict(N, weights)
80             if predActions != trueActions:
81                 numMistakes += 1
82             # Update weights
83             for action in trueActions:
84                 weights[action] -= 1
85             for action in predActions:
86                 weights[action] += 1
87         print('Iteration {}, numMistakes = {}, weights = {}'.format(t, numMistakes, weights))
88         if numMistakes == 0:
89             break
90
91 examples = generateExamples()
92 print('Training dataset:')
93 for example in examples:
94     print(' ', example)
95 structuredPerceptron(examples)

```

```

(7, ['walk', 'walk', 'tram', 'walk'])
(8, ['walk', 'tram', 'tram'])
(9, ['walk', 'tram', 'tram', 'walk'])
Traceback (most recent call last):
  File "tram.py", line 96, in <module>
    structuredPerceptron(examples)
  File "tram.py", line 84, in structuredPerceptron
    weight[action] -= 1
NameError: name 'weight' is not defined

```

```

spring2018/semilive/search2 master10h7m
python tram.py
Training dataset:
(1, [])
(2, ['walk'])
(3, ['walk', 'walk'])
(4, ['walk', 'tram'])
(5, ['walk', 'tram', 'walk'])
(6, ['walk', 'walk', 'tram'])
(7, ['walk', 'walk', 'tram', 'walk'])
(8, ['walk', 'tram', 'tram'])
(9, ['walk', 'tram', 'tram', 'walk'])
Iteration 0, numMistakes = 6, weights = {'walk': 1, 'tram': 2}
Iteration 1, numMistakes = 0, weights = {'walk': 1, 'tram': 2}

```

In this example, in the second iteration, we converged to the "true" weights. We care about the **ratio** more

Will this converge into a local optima?

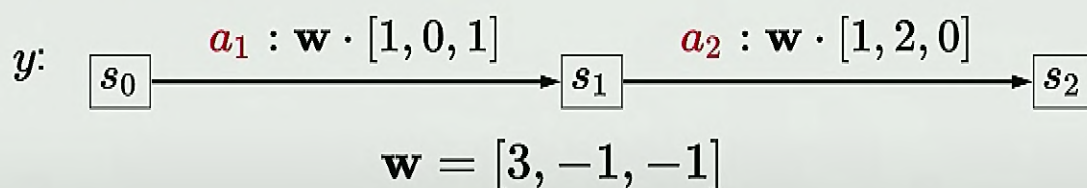
The cost function can be more complex

Generalization to features (skip)

Costs are parametrized by feature vector:

$$\text{Cost}(s, a) = \mathbf{w} \cdot \phi(s, a)$$

Example:



Path cost:

$$\text{Cost}(y) = \mathbf{w} \cdot \phi(y)$$

It can be a set of features

Applications

- Part-of-speech tagging

Fruit flies like a banana. → Noun Noun Verb Det Noun

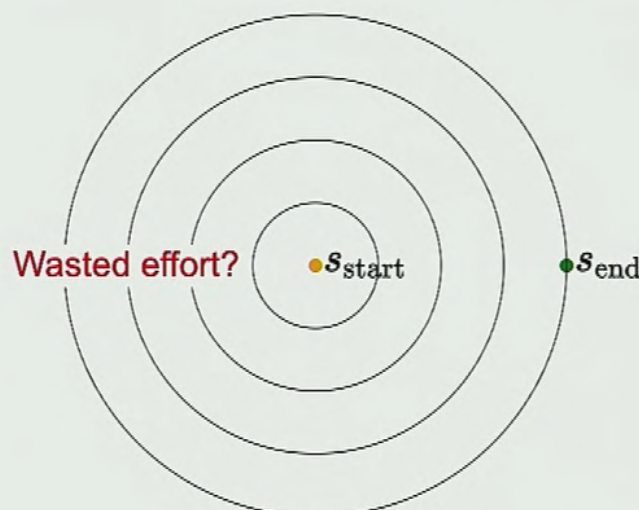
- Machine translation

la maison bleue → *the blue house*

A* Algorithm

The uniform search is uniformly exploring all the state possible The uniform cost search just explores in the order of the past cost

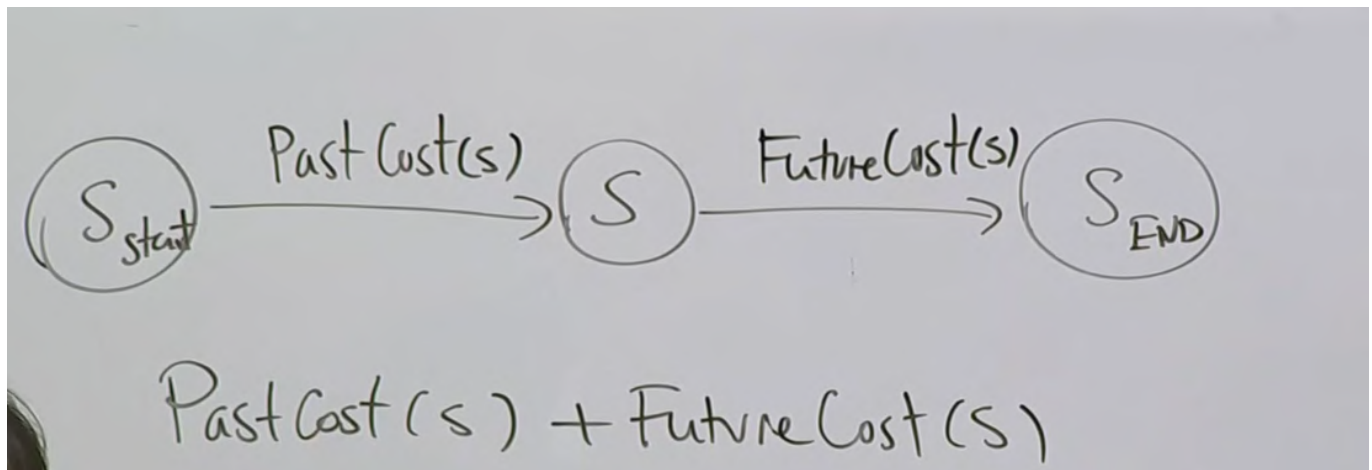
Can uniform cost search be improved?



Problem: UCS orders states by cost from s_{start} to s

Goal: take into account cost from s to s_{end}

The idea of A* is basically do the uniform cost search, but do it smarter, to move towards the goal state

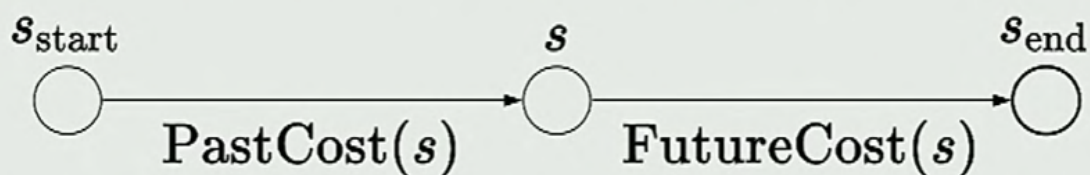


We don't really know about the feature cost, but we can estimate it as: $h(s)$

Heuristic(s) : the estimation of feature cost of s

Exploring states

UCS: explore states in order of $\text{PastCost}(s)$



Ideal: explore in order of $\text{PastCost}(s) + \text{FutureCost}(s)$

A*: explore in order of $\text{PastCost}(s) + h(s)$



Definition: Heuristic function

A heuristic $h(s)$ is any estimate of $\text{FutureCost}(s)$.

The A* basically just does uniform cost search with a new **Cost**. This will guide us to move towards the final direction

A* search



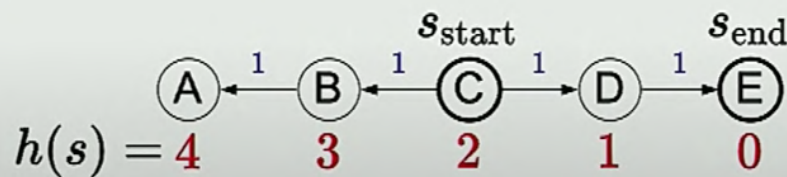
Algorithm: A* search [Hart/Nilsson/Raphael, 1968]

Run uniform cost search with **modified edge costs**:

$$\text{Cost}'(s, a) = \text{Cost}(s, a) + h(\text{Succ}(s, a)) - h(s)$$

Intuition: add a penalty for how much action a takes us away from the end state

Example:



However, the current H function is exact the future cost

Q: Is A* gonna be a greedy approach? A: Depends on the heuristic function we gonna choose, in the above example, the H function is exact the future cost, then we gonna find the most optimum one

Heuristic H

Consistent heuristics

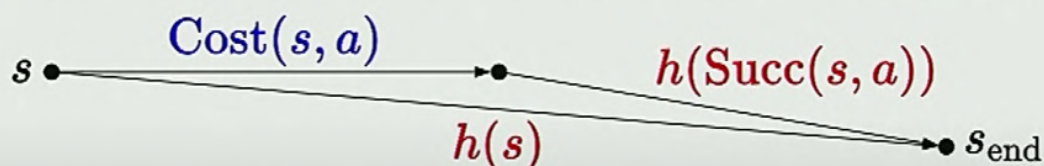


Definition: consistency

A heuristic h is **consistent** if

- $\text{Cost}'(s, a) = \text{Cost}(s, a) + h(\text{Succ}(s, a)) - h(s) \geq 0$
- $h(s_{\text{end}}) = 0$.

Condition 1: needed for UCS to work (triangle inequality).



Condition 2: $\text{FutureCost}(s_{\text{end}}) = 0$ so match it.

The Heuristic function should be consistent

- The cost should be always ≥ 0
- The cost at the end should = 0

Efficiency of A*

Efficiency of A*

**Theorem: efficiency of A***

A* explores all states s satisfying
 $\text{PastCost}(s) \leq \text{PastCost}(s_{\text{end}}) - h(s)$

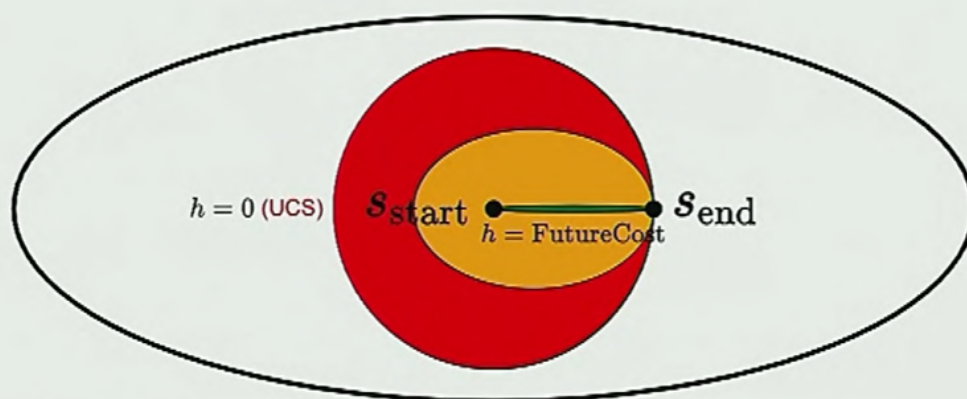
Interpretation: the larger $h(s)$, the better

Proof: A* explores all s such that

$$\text{PastCost}'(s) <$$

It is searching towards the direction So if $H(s)$ is larger(has a strong direction), then we can narrow own the future direction more effectively

Amount explored



- If $h(s) = 0$, then A* is same as UCS.
- If $h(s) = \text{FutureCost}(s)$, then A* only explores nodes on a minimum cost path.
- Usually $h(s)$ is somewhere in between.

A* is efficient

A* is Admissable

Admissibility



Definition: admissibility

A heuristic $h(s)$ is admissible if

$$h(s) \leq \text{FutureCost}(s)$$

Intuition: admissible heuristics are optimistic

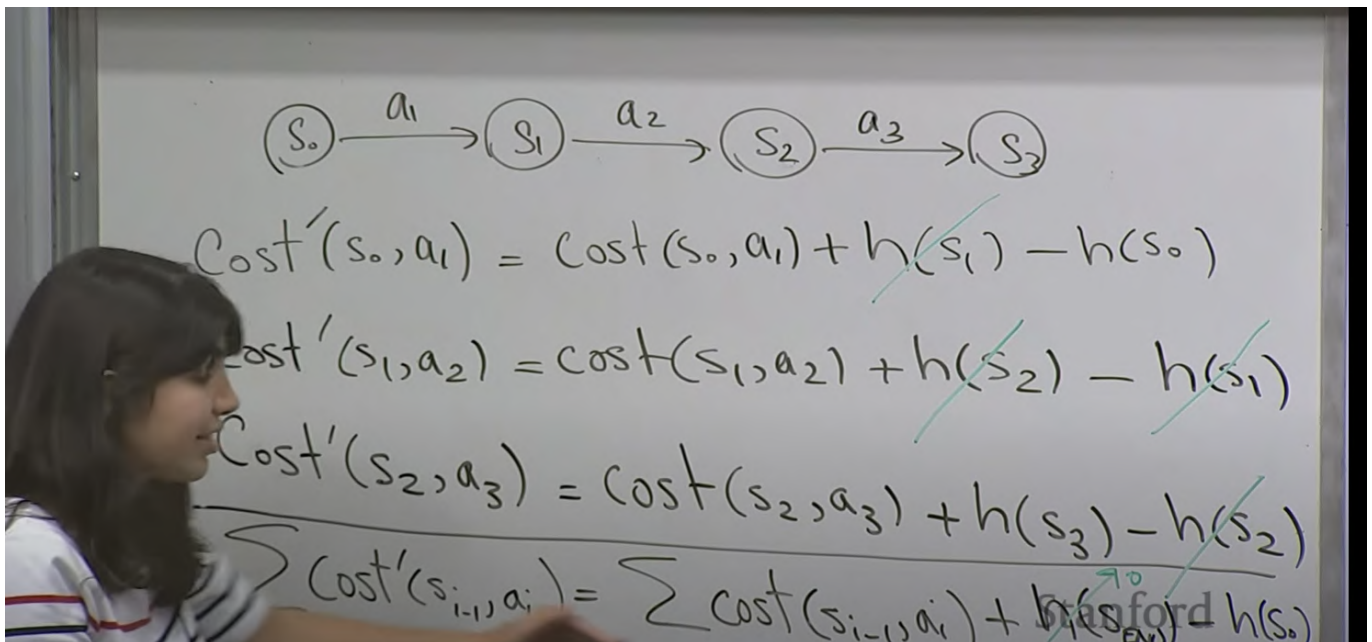


Theorem: consistency implies admissibility

If a heuristic $h(s)$ is **consistent**, then $h(s)$ is **admissible**.

Proof: use induction on $\text{FutureCost}(s)$

Mathematical proof



When you add up all things together: The cost of A* = The cost of UCS - A constant We already know that the The cost of UCS is guaranteed to be the optimal cost So even after

When we talk about the correctness, the UCS is correct, so the cost it is returning is optimal A* is just UCS with a new cost, which is the optimal cost from UCS and minus a constant So if we are optimizing the new cost, it is the same thing as optimizing the old cost So it is going to return the optimal solution

So A* is correct only if Heuristic is a constant

Correctness of A*

A* is gonna be correct IF Heuristic returns consistent

Definition

$$F = g + h$$

g = the movement cost to move from the starting point to a given square on the grid, following the path generated to get there. h = the estimated movement cost to move from that given square on the grid to the final destination. This is often referred to as the heuristic, which is nothing but a kind of smart guess. We really don't know the actual distance until we find the path, because all sorts of things can be in the way (walls, water, etc.). There can be many ways to calculate this 'h' which are discussed in the later sections.

How to calculate H

Of course you can calculate the exact h , but that would be time consuming

You can estimate h with the following approaches:

Manhattan Distance

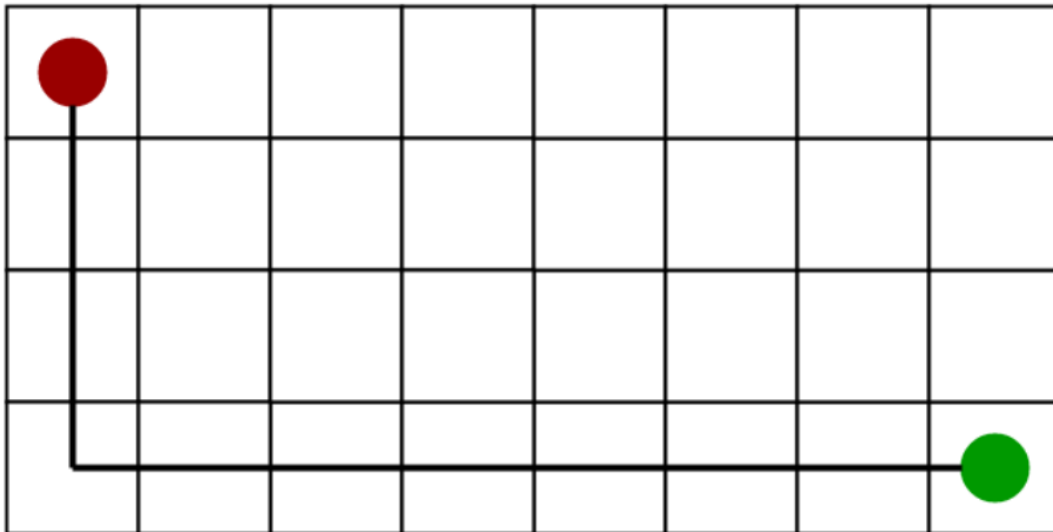
It is nothing but the sum of absolute values of differences in the goal's x and y coordinates and the current cell's x and y coordinates respectively, i.e.,

$$h = \text{abs}(\text{current_cell.x} - \text{goal.x}) + \text{abs}(\text{current_cell.y} - \text{goal.y})$$

This means: always track the distance between current point and the target point

When to use this heuristic? – When we are allowed to move only in four directions only (right, left, top, bottom)

The Manhattan Distance Heuristics is shown by the below figure (assume red spot as source cell and green spot as target cell).



Limitations

Although being the best path finding algorithm around, A* Search Algorithm doesn't produce the shortest path always, as it relies heavily on heuristics / approximations to calculate – h

Relaxation - How to calculate H

The main idea here is just relax the problem lol

Relaxation

Intuition: ideally, use $h(s) = \text{FutureCost}(s)$, but that's as hard as solving the original problem.



Key idea: relaxation

Constraints make life hard. Get rid of them.
But this is just for the heuristic!



It doesn't have to be the exact of , it can be approximation if it

Closed form solution



Example: knock down walls

Goal: move from triangle to circle



Hard



Easy

Heuristic:

$$h(s) = \text{ManhattanDistance}(s, (2, 5))$$

$$\text{e.g., } h((1, 1)) = 5$$



Easier search



Example: relaxed problem

Start state: 1

Walk action: from s to $s + 1$ (cost: 1)

Tram action: from s to $2s$ (cost: 2)

End state: n

~~Constraint: can't have more tram actions than walk actions.~~

Original state: (location, **#walk - #tram**)

Relaxed state: location

Independent subproblems

[8 puzzle]



Original problem: tiles **cannot** overlap (constraint)

Relaxed problem: tiles **can** overlap (no constraint)

Relaxed solution: 8 indep. problems, each in closed form



Key idea: independence

Relax original problem into independent subproblems.

Easier search

- Compute relaxed $\text{FutureCost}_{\text{rel}}(\text{location})$ for **each** location $(1, \dots, n)$ using dynamic programming or UCS



Example: reversed relaxed problem

Start state: n

Walk action: from s to $s - 1$ (cost: 1)

Tram action: from s to $s/2$ (cost: 2)

End state: 1

Modify UCS to compute all past costs in reversed relaxed problem

(equivalent to future costs in relaxed problem!)

- Define heuristic for original problem:

$$h((\text{location}, \# \text{walk} - \# \text{tram})) = \text{FutureCost}_{\text{rel}}(\text{location})$$

However UCS only calculates past cost To calculate future cost, you need to reverse it **That is dynamic programming all about!!!**

General framework

Removing constraints

(knock down walls, walk/tram freely, overlap pieces)



Reducing edge costs

(from ∞ to some finite cost)

Example:



Original: $\text{Cost}((1, 1), \text{East}) = \infty$

Relaxed: $\text{Cost}_{\text{rel}}((1, 1), \text{East}) = 1$

General framework



Theorem: consistency of relaxed heuristics

Suppose $h(s) = \text{FutureCost}_{\text{rel}}(s)$ for some relaxed problem P_{rel} .

Then $h(s)$ is a consistent heuristic.

Proof:

$$\begin{aligned} h(s) &\leq \text{Cost}_{\text{rel}}(s, a) + h(\text{Succ}(s, a)) \text{ [triangle inequality]} \\ &\leq \text{Cost}(s, a) + h(\text{Succ}(s, a)) \text{ [relaxation]} \end{aligned}$$

Tradeoff

Efficiency:

$h(s) = \text{FutureCost}_{\text{rel}}(s)$ must be easy to compute

Closed form, easier search, independent subproblems

Tightness:

heuristic $h(s)$ should be close to $\text{FutureCost}(s)$

Don't remove too many constraints

However if u remove too many constraints, ur heuristic is not gonna reflect future cost, so you have to find a balance

Max of two heuristics

How do we combine two heuristics?



Proposition: max heuristic

Suppose $h_1(s)$ and $h_2(s)$ are consistent.

Then $h(s) = \max\{h_1(s), h_2(s)\}$ is consistent.

Proof: exercise

Also if u have multiple relaxed heuristics, you can take the max of that, which will reflect the actual heuristics more

Code

```

from enum import Enum
import sys
from queue import PriorityQueue

sys.setrecursionlimit(100000)

Destination = 10

### Model (Search Problem)
class TransportationProblem(object):

    WALK_COST: int
    TRAM_COST: int

    WALK = "walk"
    TRAM = "tram"

    def __init__(self, destination, weights):
        # N number of blocks
        self.destination = destination
        self.WALK_COST = weights["walk"]
        self.TRAM_COST = weights["tram"]

    def startState(self) -> int:
        return 1

    def isEnd(self, state):
        return state == self.destination

    def succAndCost(self, state : int):
        """
        Return a list of (action, newState, cost) triples
        Meaning return the a list of: actions we can take, what new state we gonna
        endup at, and what the cost gonna be
        """
        result = []
        if(state + 1 <= self.destination):
            result.append((self.WALK, state + 1, self.WALK_COST))

        if(state * 2 <= self.destination):
            result.append((self.TRAM, state * 2, self.TRAM_COST))

        return result

    def printSolution(solution):
        totalCost = solution["totalCost"]
        history = solution["history"]
        print("minimum cost is {}".format(totalCost))

        for h in history:
            print(h)

# You just need to know the current state

```

```

def dynamicProgramming(problem):

    memo = {} # state -> futureCost(state) action, newState, cost
    def futureCost(state):

        if problem.isEnd(state):
            return 0

        if state in memo:
            return memo[state][0]

        minFutureCostWithAction = min(
            (curCost + futureCost(newState), action, newState, curCost)
            for action, newState, curCost in problem.succAndCost(state)
        )

        memo[state] = minFutureCostWithAction
        minFutureCost = minFutureCostWithAction[0]
        return minFutureCost

    state = problem.startState()
    minCost = futureCost(state)

    # Recover History
    history = []
    while not problem.isEnd(state):
        _, action, newState, cost= memo[state]
        history.append((action, newState, cost))
        state = newState

    return (minCost, history)

### Learning - learn the weights
#### Generate Training Examples

def predict(NumberOfBlocks, weights):
    """
    F(x)
    Input (x): N number of blocks and weights
    Output (y): path (a sequence of actions)
    """
    problem = TransportationProblem(NumberOfBlocks, weights)
    # Pridict using Dynamicprogramming or other Algo
    totalCost, history = dynamicProgramming(problem)
    return [action for action, newState, cost in history]

def generateExamples(numberOfExamples):
    trueWeights = {
        "walk": 1,
        "tram": 5
    }

```



```

    dataSet = []
    for n in range(1,numberOfExamples):
        path = predict(n, trueWeights)
        data = (n, path)
        dataSet.append(data)

    return dataSet

def structuredPerceptron(examples):
    weights = {
        "walk": 0,
        "tram": 0
    }

    for t in range(1,100):
        numberOfMistakes = 0
        for n, trueActions in examples:
            predictActions = predict(n ,weights)

            if(predictActions != trueActions):
                numberOfMistakes += 1

            # Update weights
            # In this case, we cancelled the change if two weights are the same
            # And decreased the cost that are the same with true actions
            # Increased the cost that are different from true actions
            for action in trueActions:
                weights[action] -= 1
            for action in predictActions:
                weights[action] += 1
            print('Iteration {}, numMistakes = {}, weights = {}'.format(t,
            numberOfMistakes, weights))

            if(numberOfMistakes == 0):
                break

numOfExamples = 12
examples = generateExamples(numOfExamples)
print("Training Dataset:")
for example in examples:
    print(' ', example)

structuredPerceptron(examples)

### Inference
weights = {
    "walk": 1,
    "tram": 2
}

# problem = TransportationProblem(destination=Destination, weights=weights)

```

```
# solution = dynamicProgramming(problem)
# printSolution(solution)
```