CA318
Advanced Algorithms and AI Search

Branch and Bound; Admissible Heuristics; A*

Searching Graphs/Trees (Part 2)

Branch and Bound; Admissible Heuristics; A*

Branch and Bound

- We will now refine DFS and BFS into a single algorithm general known as **Branch and Bound**
- To make this a little clearer, we will attach a cost to each of the edges in the graph
- At each iteration, we will extend **the shortest path** that can be extended
- So we will consider only **how far we have come**
  - and not how far we have left to go.

Branch and Bound Refinements

We will then refine our branch and bound in the following three ways:

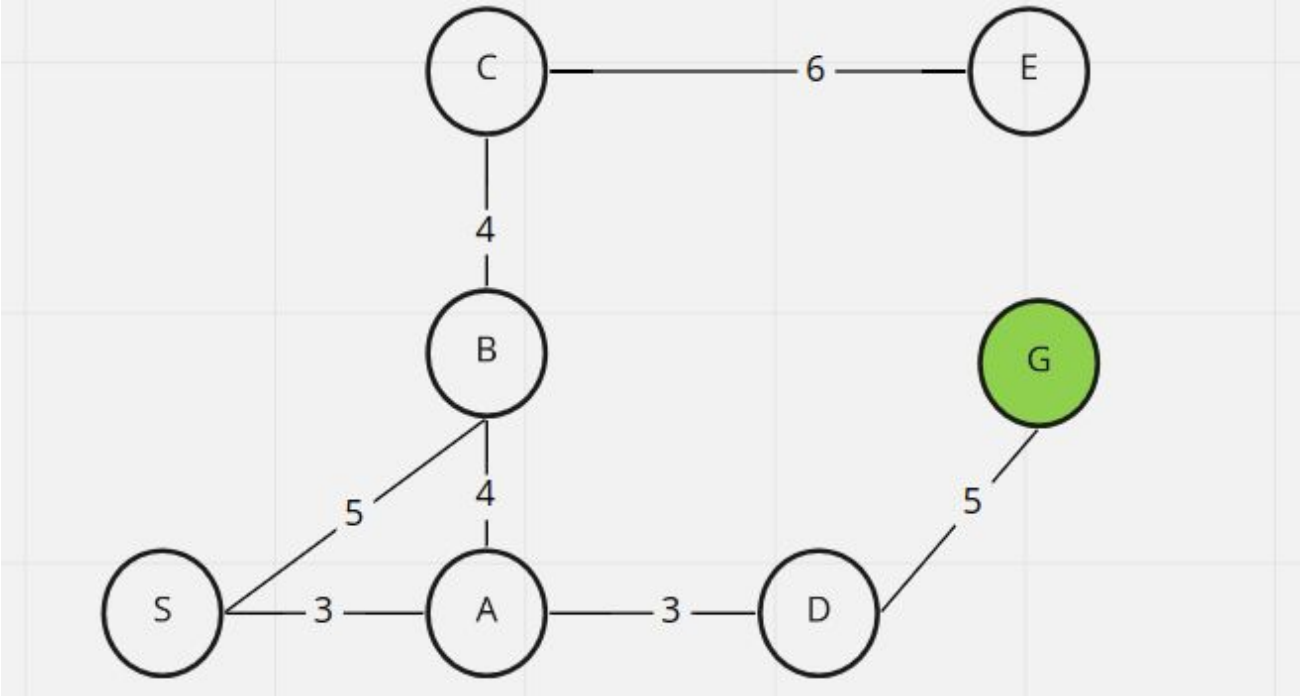
- Test for already extended nodes
- Use an admissable heuristic to estimate remaining distance
- Combine both already extended test and admissible heuristic to yield
  - A*

Naive Branch and Bound

A graph with costs

Map Search

Consider the following graph, with associated "costs". We wish to travel from S to G with the **lowest cost**

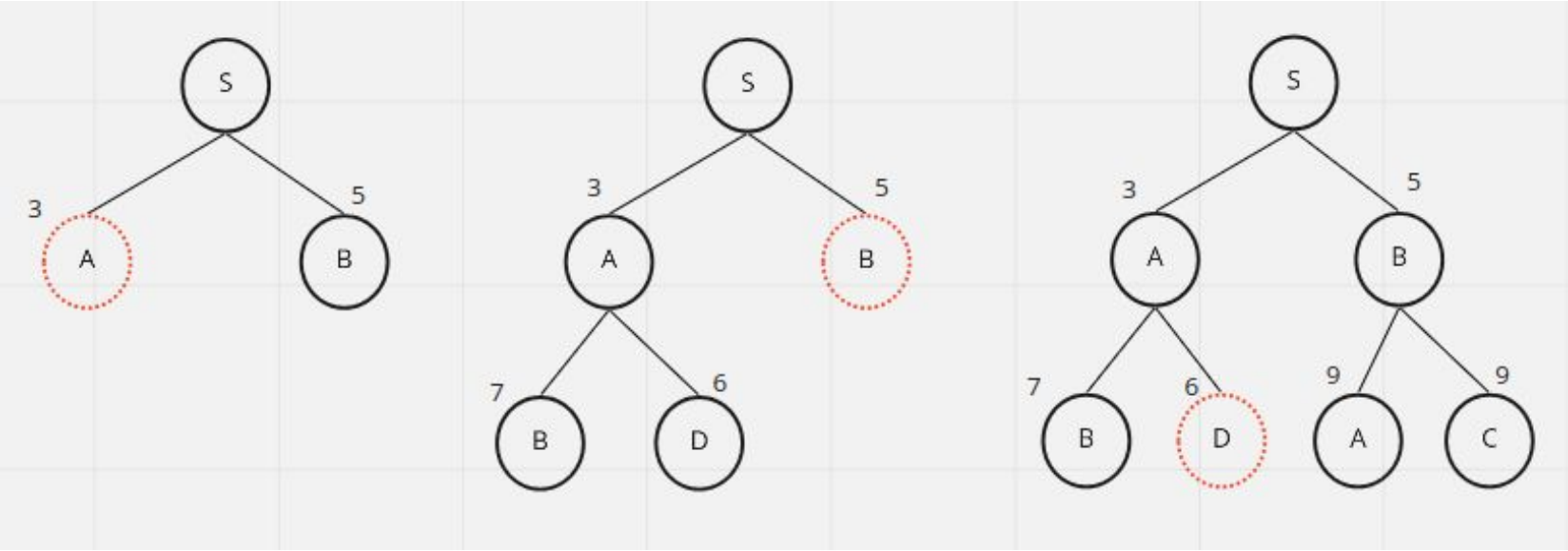


Objectives

- We want to travel from **S** to **G** with along the lowest cost route such that
  - we find the **optimal** route
  - we avoid the space complexity of BFS
  - we avoid the non-optimal solution of DFS

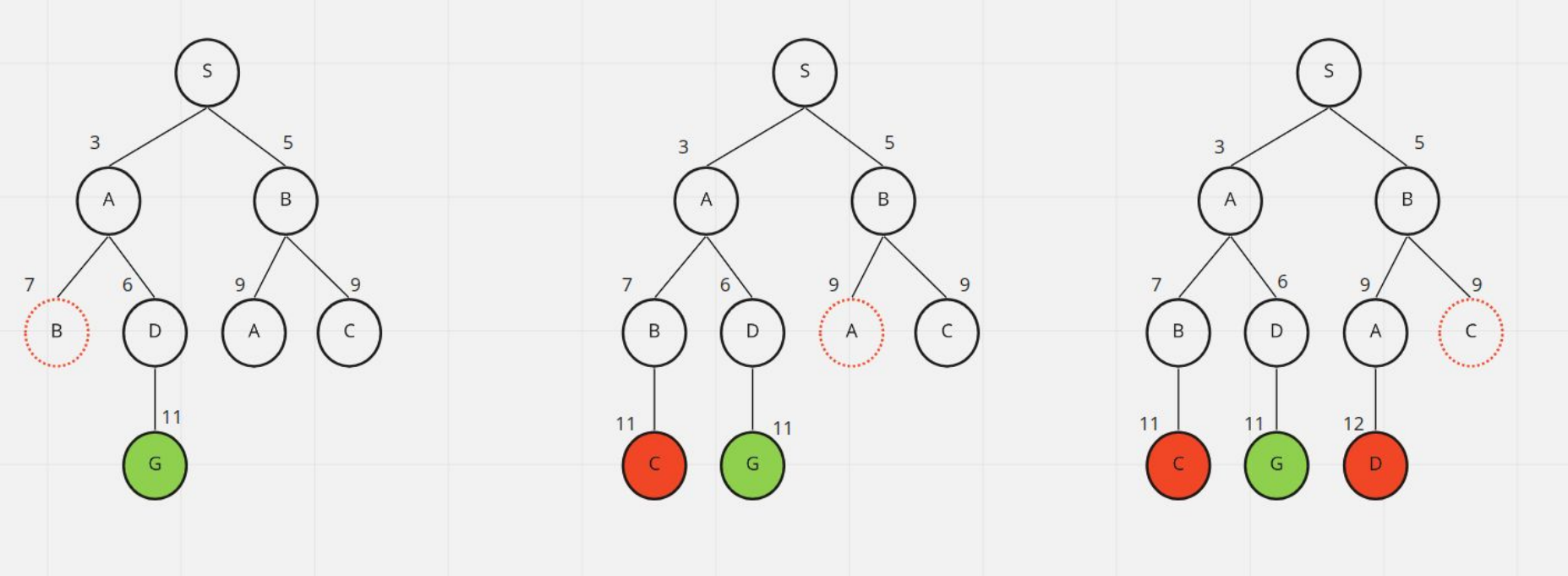
=> We will use a combination of both BFS and DFS, and use the path costs as a guide

Path Evaluation … 1



The first three node extensions

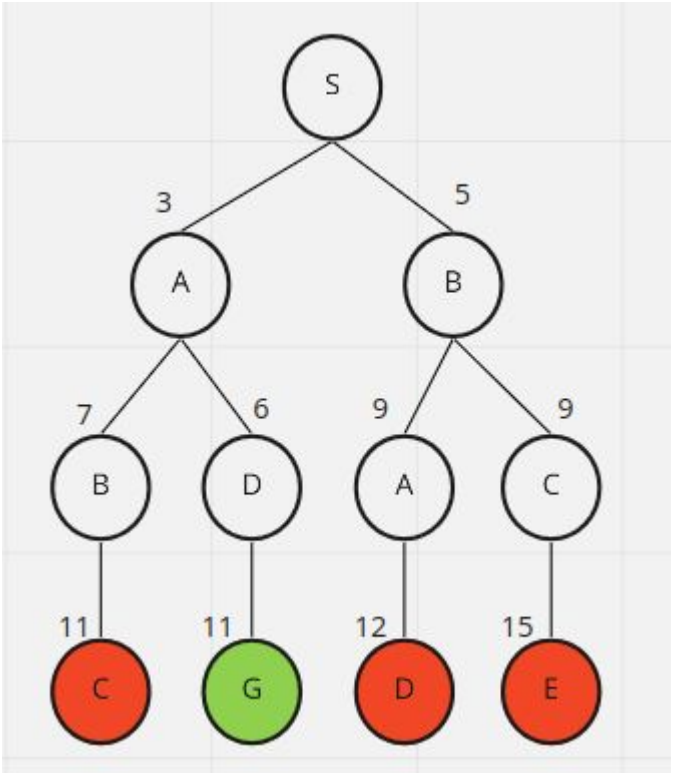
Path Evaluation … 2



The next three node extensions

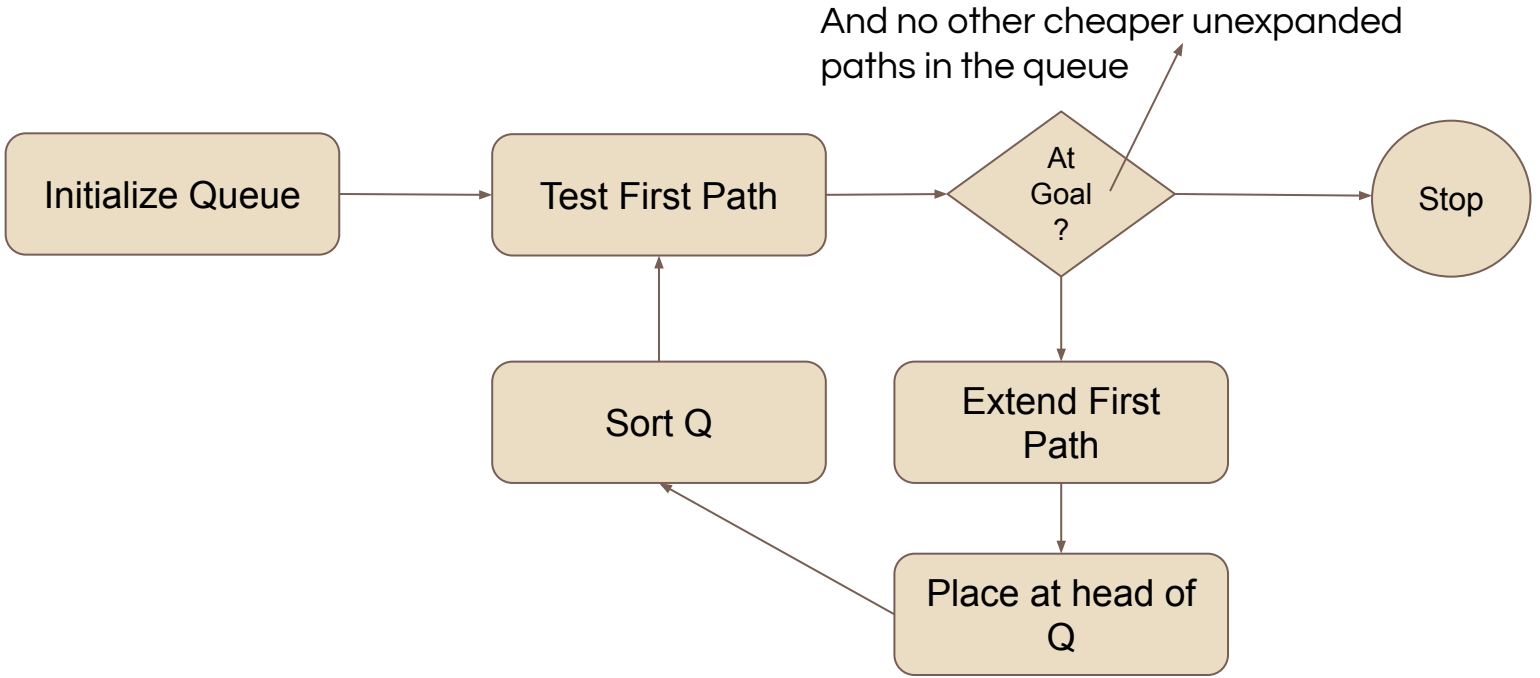
Path Evaluation … 3

Now we stop without further expansion of C, D or E**



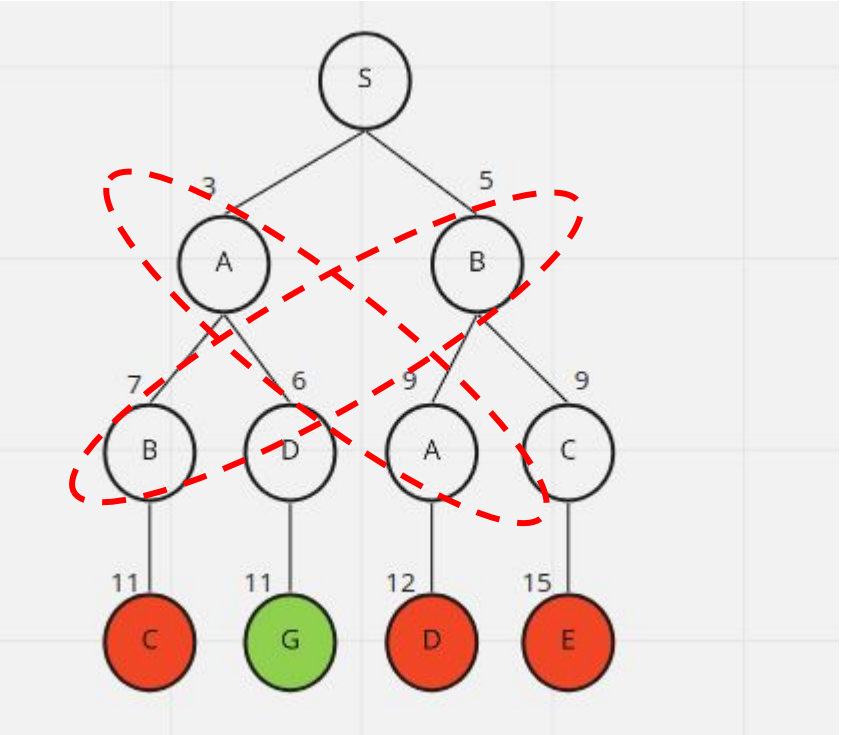
**why?

Flowchart of Branch and Bound



Naive Branch and Bound

In the previous example you can see we evaluated paths from **A** and **B** more than once



Duplication of Work

- We extended paths that went through A more than once.
- Would it ever make sense to extend A a second time?
  - No because we've already extended a path that got there with less distance.
- Will it ever make sense to extend B more than once?
  - No because we've already extended another path that gets to be by a shorter distance.

So if we keep an extended list, we can add that to branch to our advantage.

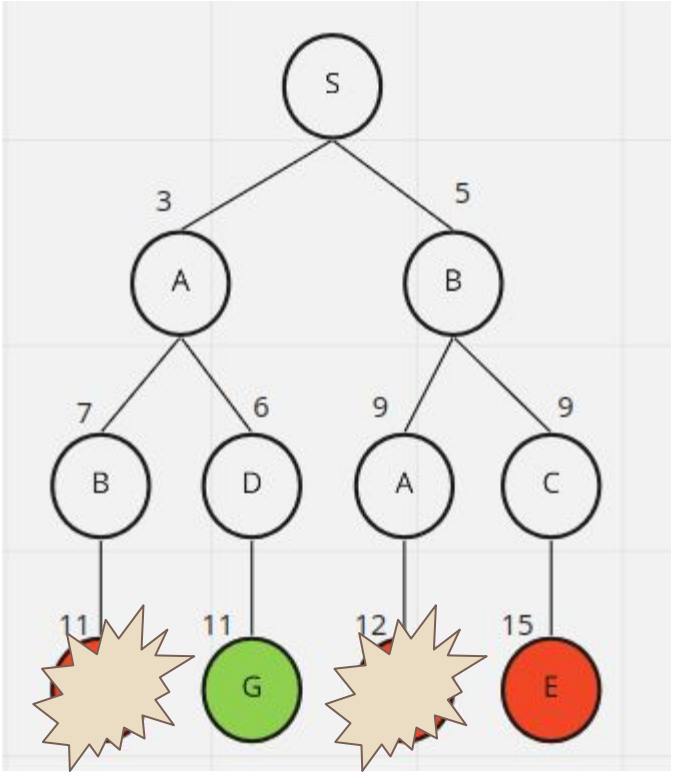
Already Extended [list]

So here we can see the end of the search graph in branch and bound.

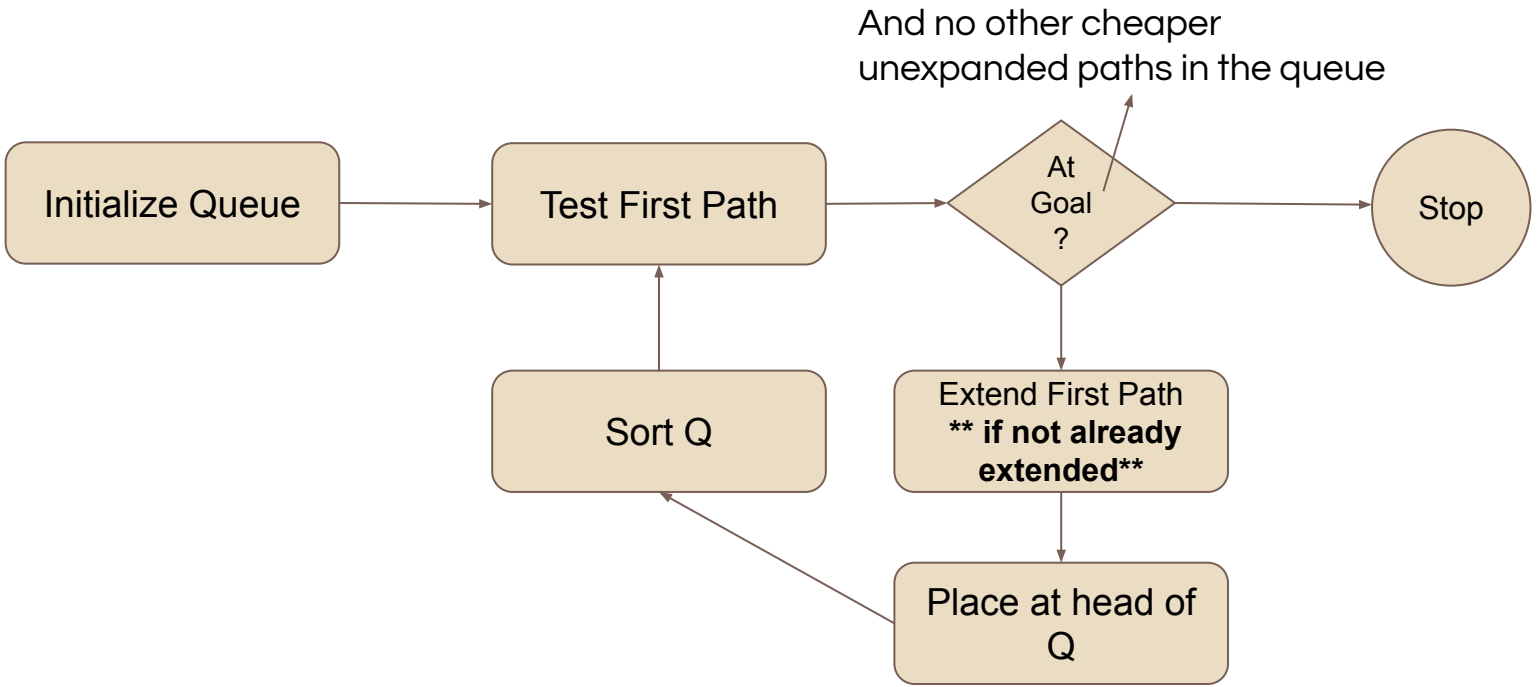
If we keep track of the already extended paths, we see both **A** and **B** have already been expanded because they were both **terminating** nodes on some previously evaluated path.

if we were to expand A a second time it could not possibly result in an shortest path

So we save some work here by using the already extended list



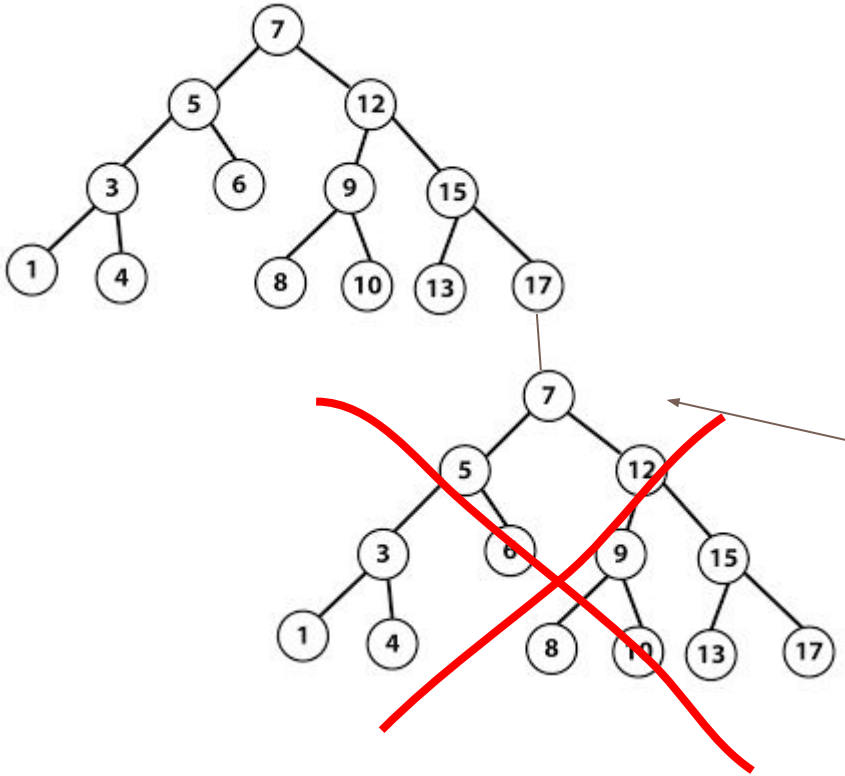
Branch and Bound with Extended List



Pruning Optimization

- If you compare the two search trees:
  - you can see that there might be vast areas of subtrees tree that are pruned away
  - and these sub-graphs don't have to be examined all.
- Here is a very compelling tree with pruned optimizations

Pruning Optimization

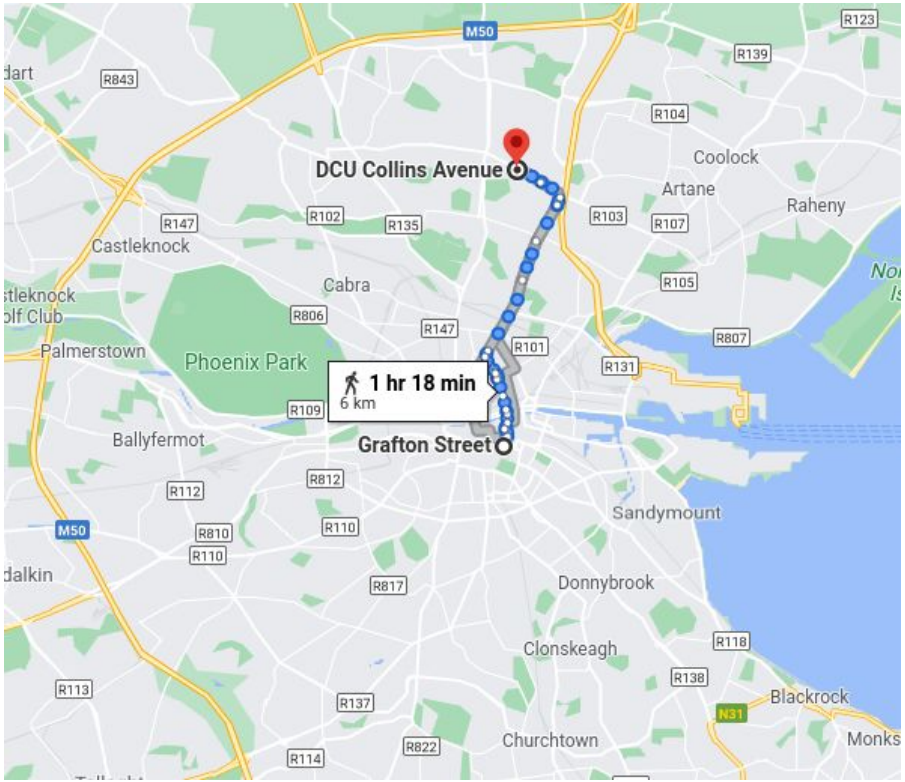


This is a contrived example, but it serves to illustrate the point - we need to improve naive branch and bound by including an already extended list

Problems Remain

Even when we include the **already extended list** concept, some obvious problems remain.

The main problem is obvious when we reorient the problem as a real-world map:



Going Away from the Goal

- The improved-naive branch and bound (as defined) has no concept of **remaining distance to the goal**
- In fact, the algorithm could spend a considerable amount of time evaluating paths **that take it away from the goal**
- This is equivalent to searching for shortest [walking] path from DCU to Grafton Street by evaluating routes through
  - Ballymun
  - IKEA / Poppintree
  - Santry
  - Malahide

Distance

- We would never do that**
- Because we intuitively have this idea of a **distance** - even if this notion is a little ambiguous
- We need a similar concept in our branch and bound
- It is called the Airline Cost, and it is a type of Admissable Heuristic
- **A Heuristic is a rule of thumb**

**would we??

Admissible Heuristics

"Airline Costs"

Admissible Heuristic

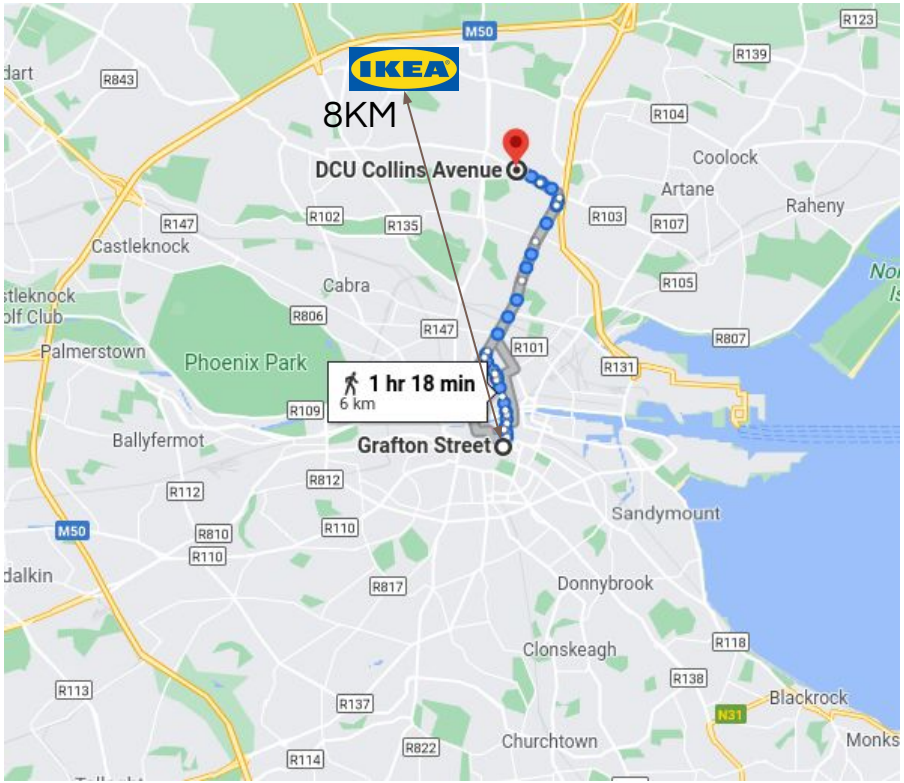
If the heuristic estimate is guaranteed to be less than the actual distance, that's called an **admissible heuristic**.

"Admissible" because you can use it for the purpose of make branch and bound more efficient.

Airline Cost

Looking again at this map we could annotate some admissible heuristics cost

This is sometimes referred to as the Airline Cost (because its a relatively straight path)

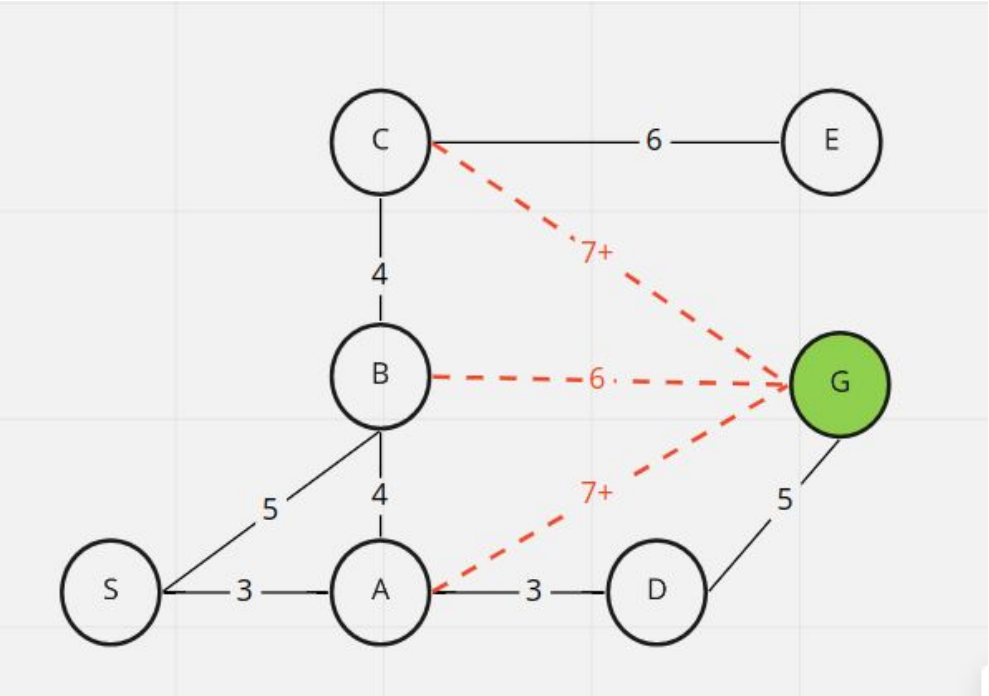


Attach an “Airline” cost

In some cases we can guess an **admissable heuristic**. In this case, a bee-line or airline distance that we attach to the graph to help with the heuristic.

Shown here in red dashed line

Lets now do this for our original problem



Airline Costs

So long as our estimates are all below the actual cost then they are admissible.

So we can create a table of these estimates now:

Node	Airline Cost
A	At least 7 (= 7^+)
B	Exactly 6
C	At least 7 (= 7^+)

Improved Naive Branch and Bound

To simplify the process, we will return to the Naive Branch and Bound **without** the use of the expanded list in order to demonstrate how it improves efficiency

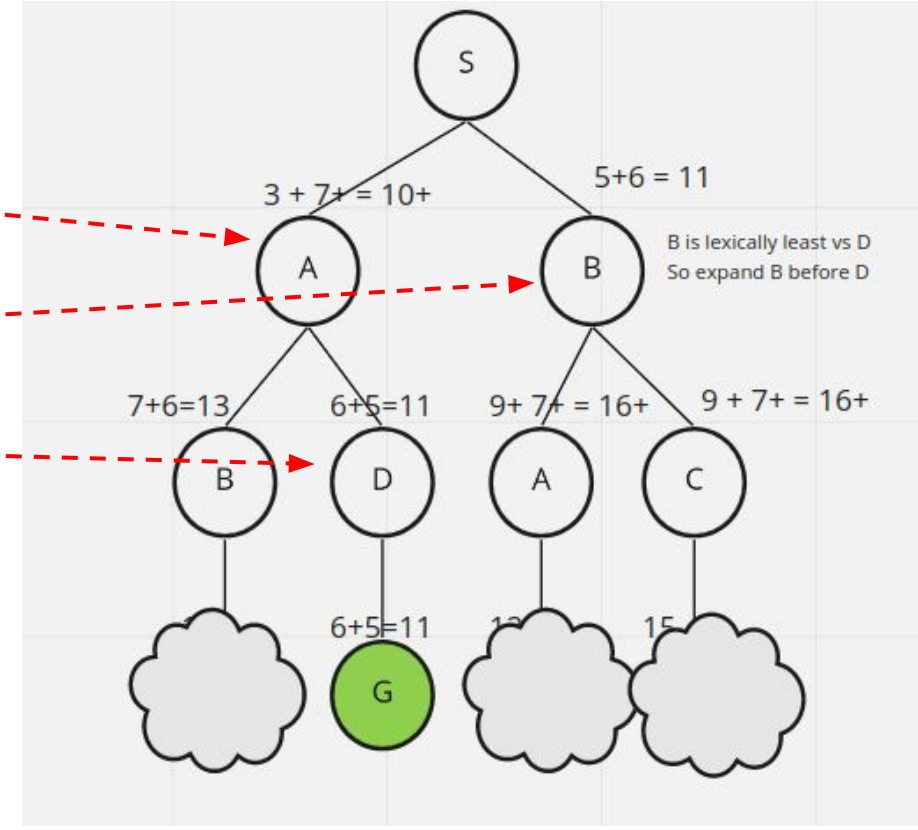
Branch and Bound with Airline Costs

1. Shortest potential path, so expand

2. Shortest potential path, so expand

3. Shortest potential path, so expand

Not evaluated



Branch and Bound with Airline Costs

- We can see that branch and bound with airline costs saves us a lot of time
- This is equivalent to walking from DCU to Grafton street, always using a street that brings us closer to Grafton street.
- So we would always want to use Branch and Bound with Admissible Heuristics
- But we can see that in the previous example, we expanded A and B unnecessarily.
- So we really do need our **Already Extended List** logic back in

Finally…

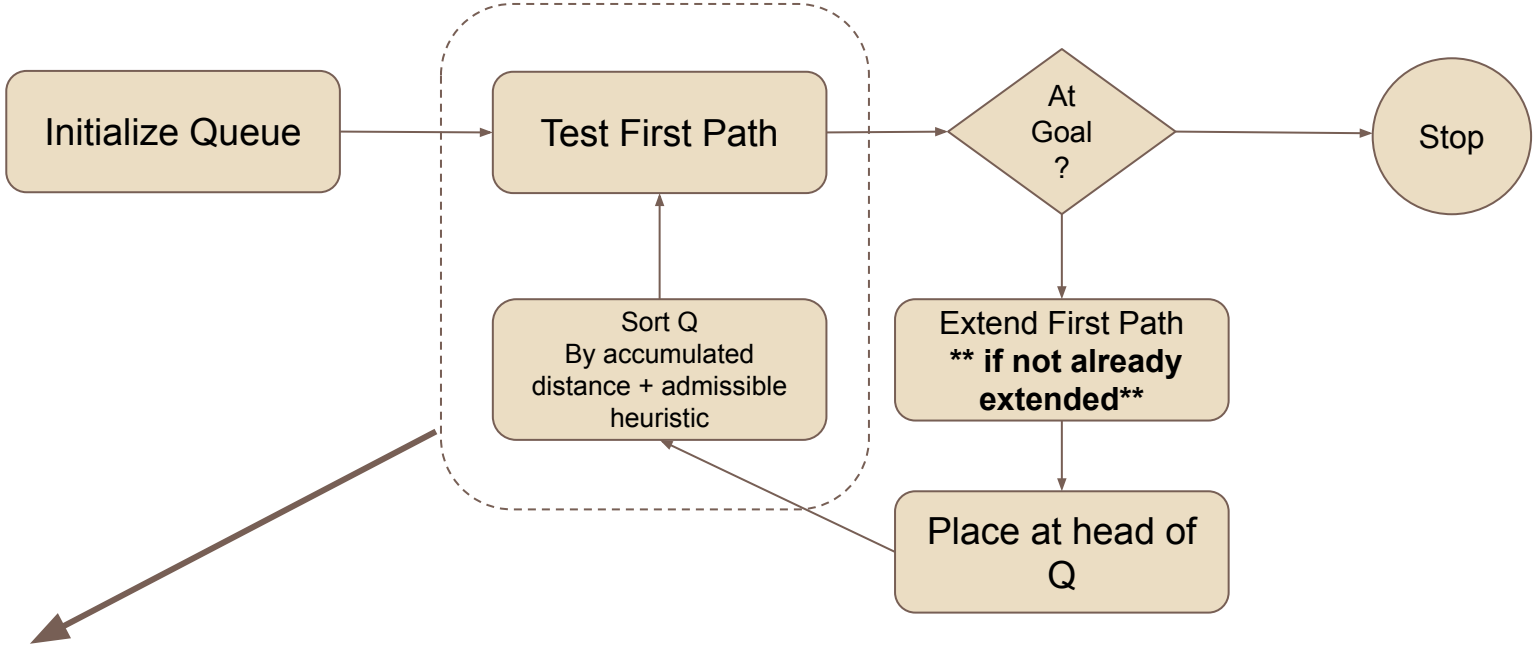
- So we now have Branch and Bound with:
  - Already Extended List
  - Admissible Heuristic

This is A*

A* Search

Branch and Bound with Already Extended and Admissible Heuristics

The New Flowchart = A*



Aside: do we really need to sort the Queue? - why might we not want to sort the Queue? What alternatives are there if we don't sort the Queue?

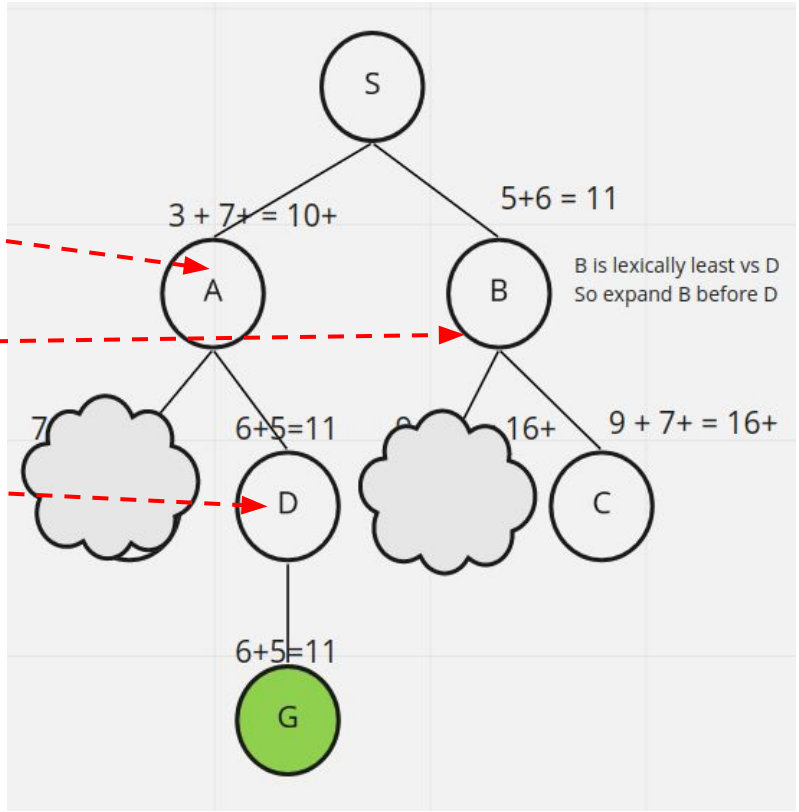
The Pruned A* Search Tree

1. Shortest potential path, so expand

2. Shortest potential path, so expand

3. Shortest potential path, so expand

Not evaluated



The Pruned A* Search Tree

We now have our A* Search tree, it finds the optimal shortest path:

- Does not expand already expanded paths
- Applies admissible heuristics

So this looks like the perfect solution.

But there is one observation here, concerning admissible heuristics.

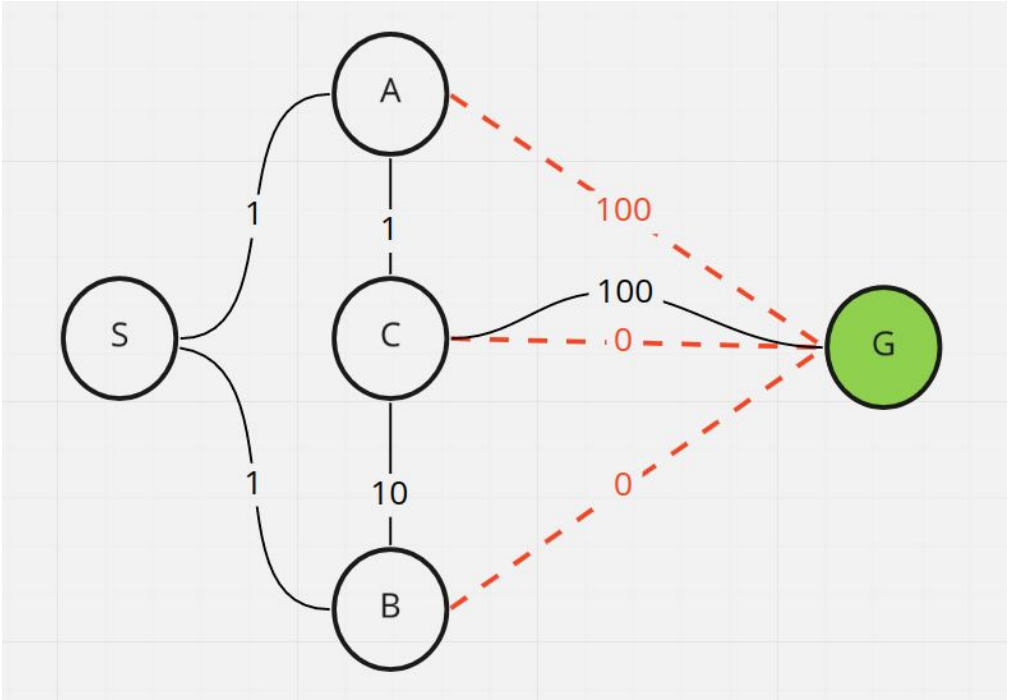
Admissible Heuristics Fail

Sometimes, the Admissible Heuristic approach can fail.

In a map path finding scenario it is reliable.

But not all search problems are map problems.

Some searches are in **non-Euclidean Spaces , eg**



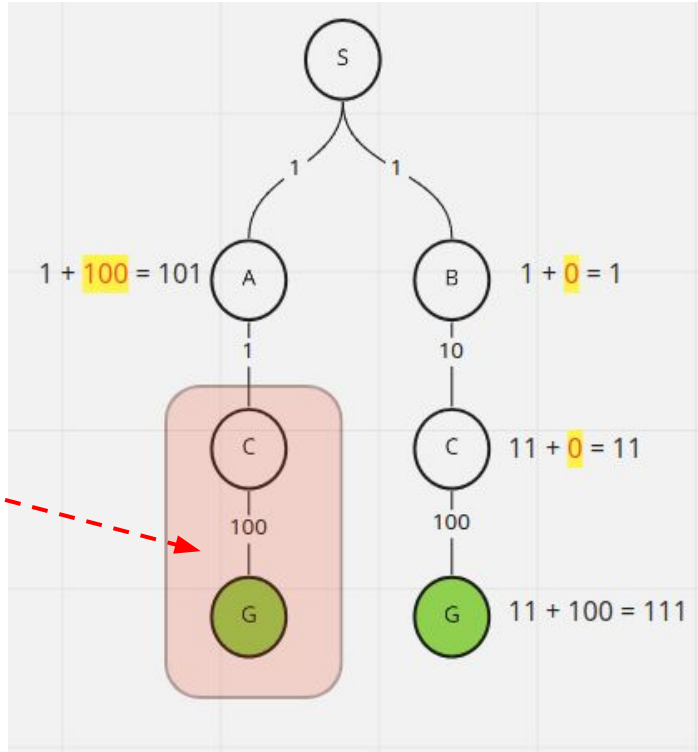
Is it acceptable to use 0? Is 0 **admissible**? Is the 100 acceptable?

A* Search Path

So obviously we have a problem here.
We are working in a non-Euclidean context, we have a path S-B-C-G=111 but a shorter path existed: S-A-C-G=102

This is an obvious situation where A* fails.

The lesson here is - A* does well with maps, but may cause problems in **non-Map** situations



Admissible and Consistent Heuristic

More Formally:

Admissible: $|H(x, G)| \leq D(x, G)$

Consistent: $|H(x, G) - H(y, G)| \leq D(x, y)$

H(A,G) = 100

H(B,G) = 0

D(A,B) = 2

| 100 | > 2 ---> **Fails Consistency**

How do we fix this?

Cost optimal A*

With an inadmissible heuristic, A* may or may not be cost-optimal.

Admissible and Consistent

- Every consistent heuristic is admissible (but not vice versa), so with a consistent heuristic, A* is cost-optimal.
- **In addition, with a consistent heuristic, the first time we reach a state it will be on an optimal path**
  - so we never have to re-add a state to the queue
  - and never have to change an entry in reached.
- But with an inconsistent heuristic, we may end up reaching the goal but in a non-optimal way

Admissible and Consistent

- These complications have led many implementers to avoid inconsistent heuristics
- However, the worst effects rarely happen in practice, and one shouldn't be afraid of inconsistent heuristics

A* Proof of Optimality

A* is cost-optimal, which we can show with a proof by contradiction.

- Suppose the optimal path has cost **C*** , but the algorithm returns a path with cost **C** > **C*** .
- Then there must be some node **n** which is on the optimal path and is unexpanded (why?)
- So then, using the notation **d*(n)** to mean the cost of the optimal path from the start to **n**, and **h*(n)** to mean the cost of the optimal path from **n** to the nearest goal then

Proof, by contradiction

$$f(n) > C^*(otherwise\ n\ would\ be\ expanded)$$
$$f(n) = d(n) + h(n)$$
$$f(n) = d(n)^* + h(n)$$
$$f(n) \leq d(n)^* + h(n)^*\ (by\ admissibility:\ \ h(n) \leq h(n)^*)$$
$$f(n) \leq C^*\ (by\ definition\ C^* = h(n)^* + h(n)^*)$$

Detour

Satisficing search

Satisficing search

- A* search has many good qualities, but it expands a lot of nodes.
- We can explore fewer nodes (taking less time and space) if we are willing to accept solutions that are suboptimal, but are "good enough"
  - what we call **satisficing** solutions
- If we allow A* search to use an inadmissible heuristic (one that may overestimate)
  - then we risk missing the optimal solution

Detour Indexes

- For example, road engineers know the concept of a detour index,
  - a multiplier applied to the straight-line distance to account for the typical curvature of roads.
- A detour index of 1.3 means that if two cities are 10 miles apart in straight-line distance, a good estimate of the best path between them is 13 miles.
- For most localities, the detour index ranges between 1.2 and 1.6

Weighted A*

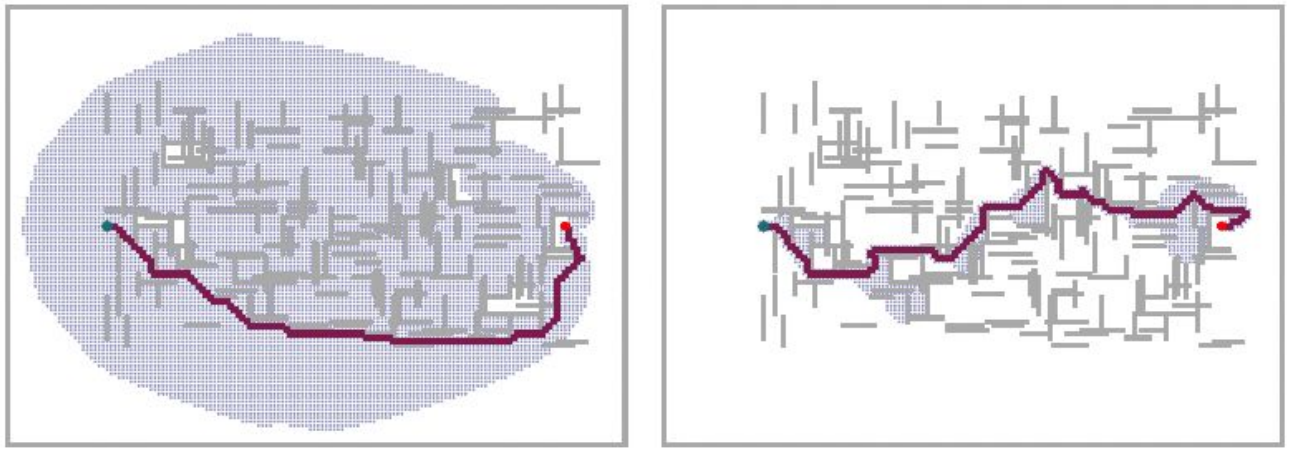
- We can apply this idea to any problem, not just ones involving roads, with an approach called **weighted A*** search
  - we weight the heuristic value more heavily, giving us the evaluation function

$$f(n) = g(n) + W \times h(n)$$

for some W > 1

Here we have a weight parameter that can be used to govern the balance between solution quality and search effort

Weighted A* Example



(a) (b)

Two searches on the same grid: (a) an A* search and (b) a weighted A* search with weight W = 2. The gray bars are obstacles, the purple line is the path from the green start to red goal, and the small dots are states that were reached by each search. On this particular problem, weighted A* explores 7 times fewer states and finds a path that is 5% more costly

Observation

- Satisficing is a decision-making strategy that aims for a **satisfactory** or **adequate result**
  - rather than the **optimal** solution.
- Instead of putting maximum exertion toward attaining the ideal outcome
  - satisficing focuses on pragmatic effort when confronted with tasks.

Thank you

Any Questions?