

Example: Missionaries and Cannibals

Lecture 2

Searching in Path Seeking problems

In this lecture we will look at the first basic technique in AI: searching.

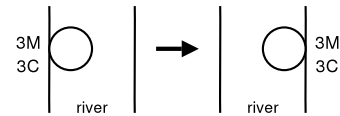
We will see that a lot of problems involves finding a path from a state to another, and will learn techniques to perform such searching efficiently.

References

- Textbook section 3.1–3.5, 4.1–4.2.

The Chess example of our last lesson is a bit too complicated, since it involves two players. Let's start with something similar.

Question: we have **3 missionaries** and **3 cannibals**, all want to cross a river with a **boat**, which can carry **one or two people**.



But missionaries are afraid of cannibals. We want to make sure that at any time, on **both** sides of the river, either there is no missionaries, or **the number of cannibals won't exceed that of missionaries**.

Note: it is still fully observable, deterministic, discrete and static.

AI(0270)

AI(0270)-2.1

Representing the world

First step: the program needs to be able to represent a "situation" of the world. We call such a situation to be a **state**.

For our problem... **how many missionaries** and **cannibals** are in each side, and **where is the boat**. E.g., in C, one possible representation:

```
struct mc_state {
    int m[2], c[2];
    int boat;
};
```

Initial state: { {3, 0}, {3, 0}, 0 }; **goal state:** { {0, 3}, {0, 3}, x } for any x.

Actually we can only reach the goal state with x=1.

But because our main focus is not programming language, we will just write a state like `state = {m: (3,0), c: (3,0), boat: 0 }`.

We will access items like `state[boat]` and `state[m][0]`.

How the world change

The program must also know how the world change: given a state and a possible operation, what is the result of taking that action. E.g.,

```
# move is in form (num_m, num_c).
# It return a new state resulting from moving num_m missionaries and
# num_c cannibals to the other side of the river
def move_people(s, move):
    s1 = s.copy() # Make a copy of the original state
    from_side = s1[boat] # Either 0 or 1
    s1[boat] = 1 - s1[boat]
    to_side = s1[boat]
    s1[m][from_side] -= move[0]
    s1[m][to_side] += move[0]
    s1[c][from_side] -= move[1]
    s1[c][to_side] += move[1]
    return s1
```

AI(0270)-2.2

AI(0270)-2.3

Successor function

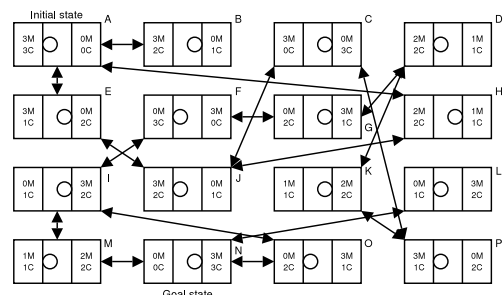
Next, given a state, our program must be able to find what are the **possible actions** and the corresponding **possible next states**. We call this the **successor function**. E.g., in our problem:

```
def next_states(s):
    ret = [] # empty list
    s1 = move_people(s, (1, 0)) # move 1M and 0 C
    if (valid(s1)): # check whether the move is allowed
        ret.append((s1, (1, 0)))
    s1 = move_people(s, (2, 0))
    if (valid(s1)):
        ret.append((s1, (2, 0)))
    s1 = move_people(s, (1, 1))
    if (valid(s1)):
        ret.append((s1, (1, 1)))
    ... # move 0M 1C
    ... # move 0M 2C.
    return ret;
```

AI(0270)-2.4

State space

Conceptually, now we a **state space**: a "graph" representing where we can go to from each of the possible state.



We want to **plan a path** from the initial state to the goal state.

AI(0270)-2.5

A few points to note...

- In real world, the state space is **too large** to generate completely, even in a computer with a lot of memory. We need to search **without generating the complete state space**.
- Instead, we start from the initial state, and **generate states** using the successor function. We might remember some or even all the state that we generate.
So the successor function defines the graph "implicitly".
- We **plans** a path. That is, when we "generate" a state, we are **not really taking the action to go to that state**. The intelligence of an AI agent comes from being able to **predict what will happen** well before actually performing the action.
- Once a search algorithm gives us a path, the agent may choose to **execute the plan** by taking the actions as suggested by the plan, in an attempt to achieve the goal.

AI(0270)-2.6

Searching: formal definition

Let's stop a while and see **how to generalize** this to many other problem. Each problem consists of:

- **States** to represent the situation in a world.
- **Successor function**: given a state, generating all <action, resulting-state> pairs.
- **Initial state**. A state where we start.
- **Goal states**. A set of states which achieves the target. This is sometimes expressed as a "criteria" (i.e., function returning true or false).
- **Path cost**. If there are many paths to the goal state, we might talk about which is better. We define path cost so that **smaller is better**.

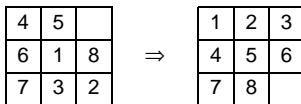
Often, path cost is the **sum of cost of actions** used to reach the goal.

We will see many such problems, so we want a **general way to solve it**.

AI(0270)-2.7

Example 2: 8-Puzzle

Example problem:



- **States**: configuration (representation: string of 9 characters).
Or any other reasonable representation, e.g., 2-D array.
- **Successor function**: the result of push-left, push-right, push-up and push-down, whichever allowed.
Number of possible actions depends on how you define them! E.g., if you define it like push-from-1, push-from-2, etc, then you'll get 8 actions. (Less is better.)
- **Initial and goal state**: defined by problem instance.
- **Path cost**: the cost of each action is 1, path cost is sum of actions.

AI(0270)-2.8

Example 3: Arithmetic puzzle

Substitute letters by distinct digits to make a formula correct. E.g.,

```
      S E N D
+     M O R E
-----
      M O N E Y
```

- **States**: a partial assignment of letters to digits
- **Successor function**: the result of adding each possible assignment. (Many!)
- **Initial state**: no letter is assigned a digit.
- **Goal**: all letters are assigned a digit, and the formula is correct.
- **Path cost**: always 0 (no preference as long as you can find an assignment).

AI(0270)-2.9

Example 4: Tic-tac-toe

- **State**: a placements of circles and crosses on the board.
- **Successor function**: the possible ways to add a circle (or cross) at certain location of the board, and their results.
- **Initial state**: empty board
- **Goal**: any state in which the game ends (i.e., one side get a line, or all board positions are taken) with ourselves winning or getting a draw.
- **Path cost**: 0 if we eventually win, 1 if we eventually get a draw.

Of course, our agent will be coded somewhat differently to cater for the fact that **there is another agent** playing with it.

This is a two-person game. We will talk about it in week 4.

For more complicated problems, searching is usually a building block of our algorithm.

AI(0270)-2.10

Search algorithms

We now have to solve the problem: **searching through the state space**.

How good we can do? It really depends on problem.

- For some really small problems (e.g., tic-tac-toe, arithmetic puzzle, 8-queen, etc), we can afford to search it exhaustively.
Search space size of all the 3 problems are less than $9! = 362880$.
- For larger problems, (e.g., 15-puzzle, rubik cube), we still want to do it exactly, but we will rely on some **additional knowledge** about the search space.
This is called "informed search".
- For yet larger problems (e.g., chess, larger variants of 15-puzzle, many real-world problems), the time and memory requirement for performing exact search is too large. We will have to do it **sub-optimally**.
E.g., losing some games, finding a path which is longer than required, etc.

AI(0270)-2.11

Criteria

We will evaluate search algorithms based on the following criteria:

- **Completeness:** does the algorithm guarantees to find a solution provided that there is one?
Completeness always depend on the amount of time and memory we have. Here completeness assume that we have sufficient memory.
- **Time cost:** how much time is required to solve a problem?
The smaller, the better. Sometimes we can tolerate a problem requiring several hours, or even several days, to solve, but that is not common. Don't it confuse with the path cost!
- **Memory cost:** how much memory is required to solve a problem?
This is much more a problem than time. We might have as much time as needed, but we probably won't have as much memory as we need.
- **Optimality:** if the algorithm does find a solution, is it guaranteed to be the best one possible?
Usually we can accept being a little bit off optimal.

AI(0270)-2.12

Branching factor, depth and cost

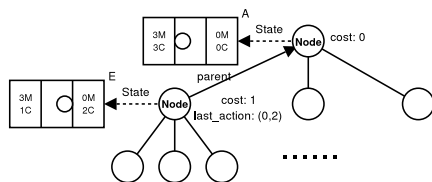
People studying AI have a rather unique notion of algorithm cost.

- If you are taking an **algorithm course**, the time and memory cost is probably a function of the number of **graph nodes** and **edges**.
- But in AI the graph is **so large that we don't want to use it as bound**.
- Example size of search space: rubik cube—reported to be 10^{17} . Chess—estimated to be 10^{120} .
- If we get only a bound the time or space requirement in terms of size of graph, we will get a simple answer: that problem is **unsolvable!**
- **Search Assumption 1:** out-degree of each state is not too large.
- **Search Assumption 2:** an answer is close enough to us.
Otherwise, we have to report "I can't find a solution".
- So we bound by two parameters, depth (d) and branch-factor (b).

AI(0270)-2.13

How to search: state and (search tree) node

- We **start from the initial state**, apply the successor function to find a next state, remember it, and repeat until we reach a goal state.
- Usually, we need to remember more information than just the state. We pack up all the relevant information, and call it a (search tree) **node**.



- When we search, we **expand** nodes, i.e., take a node, **apply successor function** on its state, **attach information additional information** to the resulting states to make them nodes, and store them.

AI(0270)-2.14

Breadth-first search

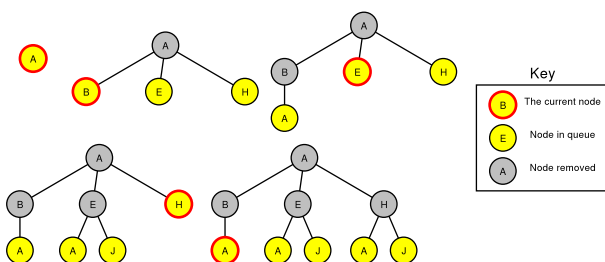
We start our tour to different searching algorithm by a very simple one.

- Breadth-first search: to search in a "**layer-by-layer**" fashion.
Seems like a good idea: closer nodes are searched before further ones.
- To do this, the algorithm would keep a **queue** of nodes.
- At the beginning, the queue simply contains a node of the **initial state**.
- In each iteration:
 - One node is **de-queued**.
 - The dequeued node is **expanded**.
 - If any expanded node has a **goal state**, the search **terminates**.
 - Otherwise, they are **en-queued** into the queue.

AI(0270)-2.15

Example

The first few steps in our Missionaries and Cannibals problem:



Note that this results into a tree (called **search tree**), **even if the state space is a graph**. There can be two nodes with the same state, though.
They might make the search algorithm slow unless properly taken care of.

AI(0270)-2.16

Code

The code of BFS is rather simple.

```
def BFS(init_state):
    # return found node
    q = queue()
    # empty queue
    q.enqueue((init_state, None, None)) # state, parent and last action
    while (not q.empty()):
        last = q.dequeue()
        for (a1, s1) in Successor(last[0]): # Expand state
            new_node = (s1, last, a1)
            if (GoalState(s1)):
                return new_node
            q.enqueue(new_node)
    return failed
```

To recover the action sequence, one will trace the parent nodes of the returned node until the root is reached, concatenating all the actions, and reverse it.

AI(0270)-2.17

BFS performance

We judge BFS with the criteria we have mentioned...

- **Complete. Optimal** if each action costs the same. How about cost?
- There is 1 node at the top level
- There are at most b nodes at the first level
- There are at most b^2 nodes at the second level
- There are at most b^i nodes at the i -th level
- So if a solution can be found at depth d , the number of nodes expanded is at most $1 + b + \dots + b^d$. This is the **time cost**.
- **Memory cost:** stores at most b^{d+1} states, each with an action list of length d .
Only the last layer is still in queue, all the others have been dequeued.

AI(0270)-2.18

What if actions cost differently?

- So what should we do if we want optimality but **action costs are not all the same?**
- We **modify the nodes** so that it also contains the path cost.
- We **modify BFS**: instead of a queue, we use a **priority queue**, with priority defined to be inverse of the path cost.
So that lower cost implies higher priority and thus get dequeued earlier.
- We will also have to make sure that the function checks for goal state **only when the corresponding node is about to expand**.
Not when another node expands resulting into the corresponding node. At that time it is still possible that another "cheaper" node is about to be created.
- Memory and CPU costs now depend on the number of nodes that **costs less than the least cost goal node**.
Multiplied by the branching factor, since the bottom layer contains nodes that are more expensive than the path cost.

AI(0270)-2.19

Code of Uniform-cost search

So Uniform-Cost Search looks like this...

```
def UCS(init_state):
    q = priority_queue()
    q.enqueue((init_state, None, None))
    while (not q.empty()):
        last = q.dequeue()
        if (GoalState(last[0])):
            return last
        for (a1, s1) in Successor(last[0]):
            q.enqueue((s1, last, a1))
    return failed
```

Note the similarity with BFS. Indeed, many other algorithm is exactly the same, apart from using different data structure. In general we call such algorithms **TreeSearch**.

AI(0270)-2.20

How bad it is really?

The depth and branching-factor determines how difficult a problem is.

- Missionaries and cannibals: branching factor close to 1, depth 11.
- 8-puzzle: branching factor between 2 to 3. Depth around 20.
Actually search space is very small here: around 9!. Searching everything here is still possible.
- 15-puzzle: branching factor between 2 to 3. Depth around 30.
- **Seems not good...** Where to find the 1000T byte memory needed?!
But then it's natural that it is difficult: such block pushing problems are NP-hard.
- Rubik cube? Branching factor around 13 to 14, depth around 20.
This is much worse, and we really need better idea than just searching blindly.
- Chess: branching factor around 35, and depth around 40 (usually).
Much too difficult to do any variant of BFS on it. But don't expect too much: the problem is P-SPACE hard (much worse than NP-hard).

AI(0270)-2.21

Depth-first search

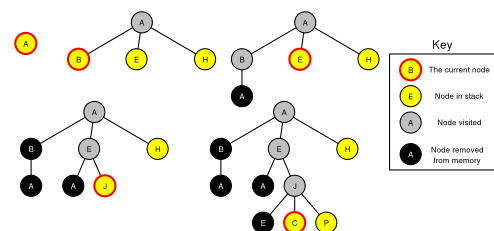
If it is not good doing it breadth-first, how about depth-first?

- **Idea:** consider one expanded nodes and **everywhere it can go**, before considering the next node.
- **Implementation:** Just use a stack in *TreeSearch*.
- **How good is it?** Not very good, unfortunately. The primary problem is that it might get stuck in an unrelated branch.
 - **Complete** only if the **tree is finite. Not optimal.**
BFS doesn't need a finite tree. As long as branching factor is finite it's okay.
 - **Time cost:** $O(b^m)$ if tree is of height m .
Or actually, the size of tree. You can't bound by branching factor and depth.
 - **Memory cost:** $O(bm)$. (Big improvement over BFS!)

AI(0270)-2.22

Example

If we search the Missionaries and Cannibals state space using DFS, the first few steps looks like this...



This requires that we avoid repeated states. Otherwise, the DFS will not throw away the path from A to B and then back to A, and will continue to go to B, A, B, ... infinitely.
We will learn how to deal with repeated states soon.

AI(0270)-2.23

Back-tracking search

- So far we concentrate on a successor function that **return all “next states” at once**. That is a good thing for *TreeSearch*.
- But if we can have the next state one-by-one, we can **further reduce** the memory requirement. The idea is to generate a state **only when needed**.
- E.g., in the last page, nodes B, E and H are generated on the first expansion. This could be a problem if the branching factor is larger.
- To save memory, we can generate **only B** at the beginning, and generate E and H only when the whole subtree of B has no solution found.
- One popular implementation: use **recursion**. When the search for subtree B failed, A receives control again and consider E.
- So control goes **back** to A when B fails, and we call it **backtracking**. Accordingly, this variant of DFS is called backtracking search.

AI(0270)-2.24

Code

- This require us to generate B and C at different time, so successor function is not suitable.
- Instead, let's assume we can create an object using *GenSuccessors*, which will generate the successors one after another.

```
def RecursiveDFS(parent): # initially parent=(init_state, None, None)
    if GoalState(parent[0]):
        return parent
    for (action, state) in GenSuccessors(parent[0]):
        result = RecursiveDFS((state, parent, action))
        if result != failed:
            return result
    return failed
```

There is really a support for such kind of generator objects in Python, but too bad that C++ doesn't have such support. In C++, one would instead embed the DFS code into the state generating code.

AI(0270)-2.25

Dealing with loops

As we have seen, **state spaces of most AI problems are not trees**.

- **BFS: some states will have more than one node**, ending up with more nodes than it requires. But the algorithm will **still terminate**.
- **DFS: it just get stuck** (in a search tree of infinite depth).
So the algorithm would never terminate until it bail out using up all memory.

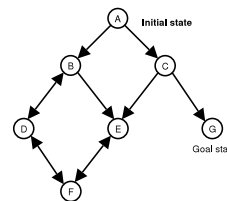
How to deal with that?

- Do not return to the state you **just came from**.
Minimal, generally a must in graph search.
- Avoid repeating any state in **any single path**.
This requires remembering the **path** with the states.
- Record all **seen states**, and refuse to create nodes with such states.
This might require too much memory, though.

AI(0270)-2.26

Example

Suppose DFS is used to search the following state space:



No checking: A, B, D, B, D, ...

Don't go back to where you come from: A, B, D, F, E, E, F, D, B, E, F, D, B, ...
One step checking don't buy you that much.

Don't go back to anywhere in path: A, B, D, F, E, E, F, D, C, E, F, D, B, G.
E.g., when you descend from A towards B, E, F, D and then B again, ignore the new B.

Don't go to expanded states: A, B, D, F, E, C, G.

AI(0270)-2.27

Don't go to expanded states: implementation

In order to implement the “don't go to expanded states”, one would use a set to keep track of the found states:

```
def GraphSearch(init_state):
    q = priority_queue()
    q.enqueue((init_state, None, None))
    closed = set() # empty set
    while (not q.empty()):
        last = q.dequeue()
        if (GoalState(last[0])):
            return last
        if not last[0] in closed:
            closed.insert(last[0])
            for (a1, s1) in Successor(last[0]):
                q.enqueue((s1, last, a1))
    return failed
```

Sometimes repeated states are still generated in the queue, although it won't be visited twice. If we are doing BFS rather than UCS, we can change the place we check *closed* and insert elements there to prevent this.

AI(0270)-2.28

Iterative deepening

- So **BFS has all the good properties** except that it eats all our memory, while **DFS has much more reasonable memory usage** but all the good properties of BFS go away.
- It is possible to **combine the good properties** of both algorithms!
- The idea: if we know the depth *d* of the best solution and **limit our search depth of DFS to d**, the nodes expanded are exactly those expanded in BFS before the *d*-th iteration.
- So we can actually **achieve the effect of BFS using DFS**, thus reducing the memory requirement exponentially.
- But we don't know *d*! Simple: **try it one by one!**
- The algorithm is called **Iterative Deepening DFS (IDS)**. It simply repeatedly invoke DFS, with a depth limit increasing from 1 until search succeed.

AI(0270)-2.29

Code

Once we know the idea, the code is more or less trivial.

```
def IDS(init_state):
    i = 0
    while True:
        result = DFS_with_depth_limit(init_state, i)
        if result != failed:
            return result
        i += 1
```

How to implement DFS_with_depth_limit? Just the same as DFS, except that one will also record the depth in the node: **if the depth is larger than the limit**, don't push its onto the stack.

So once all shorter paths are tried, "failed" is returned, so that we can start a new round with a larger depth limit.

How good is it?

At a first glance, repeating everything done by previous iterations of IDS seems stupid. But it is not.

Reason: **the cost to search the last level usually dominates the cost.** That is, searching in the previous levels are essentially free.

IDS is

- **Complete.**
- **Optimal** if all actions cost the same.
- **Time cost:** $O(1 + (1 + b) + (1 + b + b^2) + \dots + (1 + b + \dots + b^d))$
- **Memory cost:** $O(bd)$
By being a little less aggressive in minimizing time cost, we get back a lot of memory. But the memory might be needed to keep track of repeat states anyway.

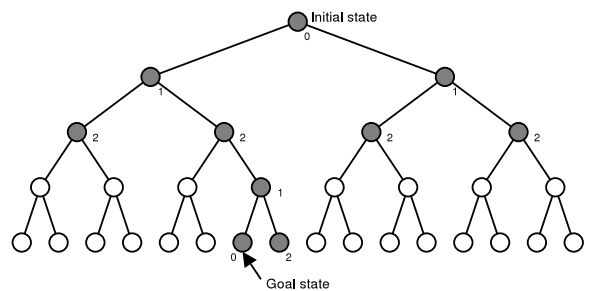
Bidirectional search

What if we have much more memory? Can we use it to search faster?

- One possibility: to **use BFS from both sides.**
- That is, we call BFS **both** for the **initial state** and the **goal state.**
- In each iteration, we will **expand one layer from each side.**
- Search terminates if **some state is expanded from both sides.**
How to check this? Keep all states generated from one side in a hash table.
- How good? It is still **complete**, and still **optimal.**
- **Time cost:** $O(b^{d/2})$
- **Memory cost:** $O(b^{d/2})$
- So we can search **twice as far** as BFS!

Example

Let's see what happen on a SS of full binary tree with 5 levels.



Shaded nodes are expanded by bidirectional search. The numbers are the depths from the initial and goal state. Note that BFS might expand all states.

Applicability

You can use bi-directional search only if:

- We have an **explicit goal state** (or just a few).
In other words, you can't do bi-directional search if you only get a goal checker. You must have a state to start with.
- We can work **backwards.**
Given a state, we know all states which can reach it.
- We are **not really that far** from the goal state.
We can only reach twice as far as a BFS.
- We have **a lot of memory** to play with.

For these reasons, bi-directional search is **not very frequently used.**

To really reduce the memory and time requirements, we will need some knowledge about the domain. We will **examine them next.**

Importance of domain knowledge

- All **un-informed search algorithm** that we've seen share a property: it **needs time in the order $O(b^d)$, i.e., exponential.**
Seems very bad, but we can easily show that this is the best we can do if the graph is arbitrary and we want a complete and optimal algorithm.
- The problem is that we **throw away everything we know about the problem**, pretending that we **know nothing about the state-space.**
- In reality, we usually get some information about the state-space, only that they are **rather inaccurate.**
On the other hand, if we have exact knowledge of the structure of the SS, we probably want to use a more specialized program rather than search through it arbitrarily.
- Also, we want our method to be kept **general** enough to solve a **wide range of problems.**

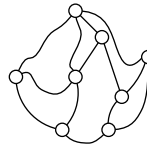
Heuristic function

- So we want a **general way to express domain knowledge** (i.e., properties of all SS that comes from the same problem) in **searching**.
- In this lecture we discuss **the simplest way to describe domain knowledge**: a function to describe how **"good"** is a state.
Later, under the topic of "planning", we will see a completely different way to express domain knowledge, leading to completely different algorithms.
- What is meant by good? Let's have a simple definition: an estimate of **the path cost to move from the current point to the goal state**.
- We assume that **path cost is the sum of action costs**.
Luckily, this still covers many problems.
- Any such function is called a **heuristic function**, or a **heuristic**.
In most other cases, heuristic means "rule-of-thumb" that improves average case performance of an algorithm.

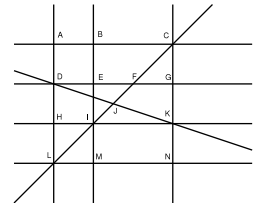
AI(0270)-2.36

Examples

Let's see why **heuristic functions encodes domain knowledge**:



Possible heuristic: geometric distance.



Possible heuristic: North-South distance + East-West distance.

Note that geometric distance is **still a valid estimate** on the right, but we can improve the estimate because we know it's mostly **rectilinear**.

The street map on the right is somewhat like that of Manhattan, and the heuristic is called the **Manhattan distance**.

AI(0270)-2.37

What is a good heuristic?

For a function to serve as a heuristic, we only require two things:

- The function **returns 0 at all goal states**.
In other words, the minimum path cost to go from a goal state to any goal state is 0—do no action.
- The function always returns a **non-negative real number**.
In other words, there is no negative action cost.

But to be a good estimate, we want the estimate to **accurately reflect the required cost**, and to be **reasonably easy to compute**.

A **special property**: if an estimate **never overestimate the path cost**, it is called **admissible**.

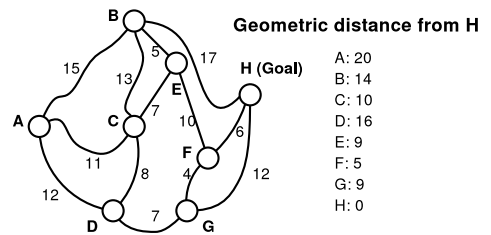
We will see that only admissible heuristics can be used in a complete and optimal algorithm called A* search.

A simple example: the function that always return 0 is an admissible heuristic, although it is a poor one (since it reflects no domain knowledge).

AI(0270)-2.38

Let's be more concrete...

On the left is the state space, on the right is a heuristic function.

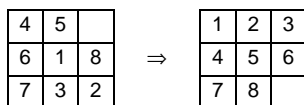


So far, to deal with such a state space, all we can do is to apply uniform cost search and ignore the heuristic (even though it is quite accurate). We will see how to do better.

AI(0270)-2.39

More examples of admissible heuristic

Let's see the block-pushing problem, **8-puzzle**. Consider the following problem instance:



- **Heuristic 1: number of wrong pieces.** The heuristic function is the number pieces in the wrong place. Here, 7 (pieces 1, 2, 3, 4, 5, 6, 8).

Clearly admissible: any wrong piece needs at least one "push" itself.

- **Heuristic 2: sum of Manhattan distances.** The heuristic function is the sum of the Manhattan distances between the initial and goal position of each piece. Here, $2+3+3+1+1+2+0+2=14$.

Admissible? Better or worse than heuristic 1? Why?

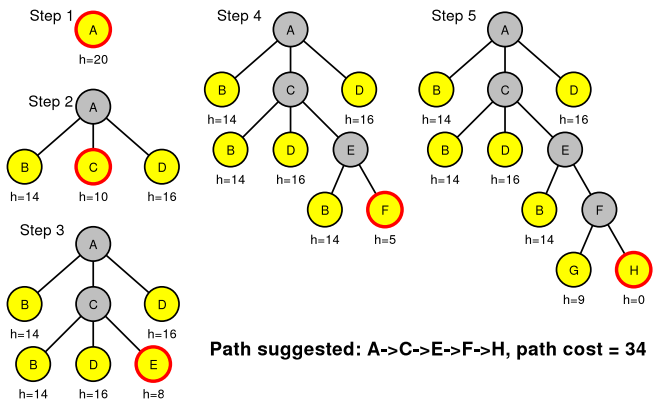
AI(0270)-2.40

Greedy Best-first search

- Given that heuristic function ("**h-value**") represents how far is a state from a goal state, it is natural to **explore low h-value states first**
- How about **slightly modifying one of these algorithm**, e.g. BFS, to do this?
- **Re-organize the data structure** once again, this time we use a **priority queue** with the priority being the **inverse of the h-value**.
The algorithm is really very flexible. We will see one more ways to prioritize the states in the queue.
- That is, everytime we find something to expand, we choose the one with the **lowest h-value**.
- This is called **greedy best-first search**.

AI(0270)-2.41

Example application: A to H...



AI(0270)-2.42

Performance

Note the similarity of greedy search and DFS: both tries to **follow a single path all the way to the goal**, until it finds a dead-end and at that time it **backtrack**, i.e., try again at expanded nodes that are not yet explored.

As a result, it shares the defects of DFS:

- It is **complete only in state spaces that are finite**. In infinite state spaces, it can get stuck in a wrong subtree infinitely long.
- It is **not optimal**, because it might get to a sub-optimal tree and find a solution there.
- **Time cost**: size of search trees when fully expanded.

But greedy search also has its unique problems:

- **Memory cost**: size of search trees!
Worse than BFS and DFS. That happens when heuristic leads us to stay at wrong side of a tree for long. Greedy Best-first search usually do well, though.

AI(0270)-2.43

A easy fix

Can we improve the situation?

- The greedy strategy has a **problem that is easy to fix**: when it gets a state from the priority queue, it gets the one with the **minimum h-value**.
- But this doesn't reveal the real cost: **the real cost also includes the cost to go from the initial state to that state (called the g-value)**!
- So the fix is easy: to use the inverse of **the sum of g-value and h-value of the state** as the priority function.
- The sum of g-value and h-value is called the **f-value** of the state.
- How good is the result? That depends on **how good is the heuristic**. In worst case, still as bad as Greedy Best-First Search.
Imagine that the first step of the correct path has a **huge** heuristic, while every other path has a small one which doesn't reach the goal.

AI(0270)-2.44

Special case: admissible heuristic and A* search

But there is an important **special case**:

When the heuristic h is admissible, the above algorithm is optimal, and is complete on most graphs.

Be reminded that admissible functions are those that never overestimate the path cost to goal state, although it is allowed to underestimate.

In this case, **the search algorithm is called A* search**.

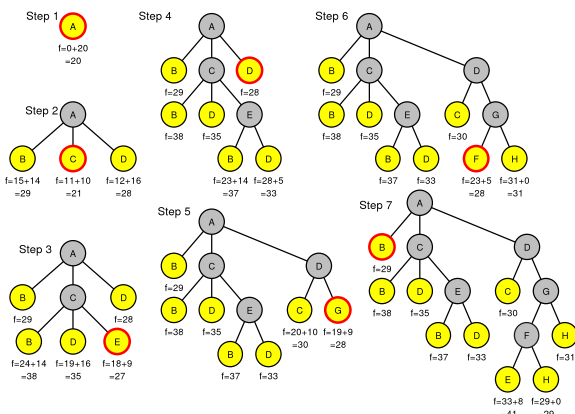
Let's use the map as an example again.

Here, the **geometric distance to the goal is an admissible heuristic**, because of triangle inequality.

Suppose we want to go from A to H...

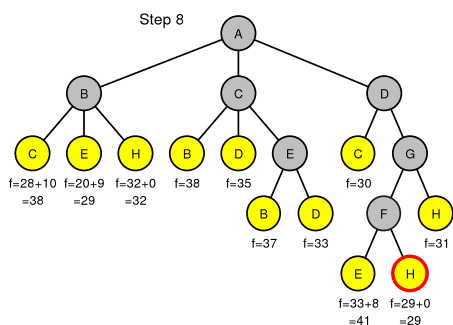
AI(0270)-2.45

Example: A* search



AI(0270)-2.46

Example: A* search (cont'd)



Path suggested:
A->D->G->F->H
Path cost = 29

AI(0270)-2.47

Why it is optimal?

- Now the question: **why is A* search optimal?**
- The answer lies in a property of the f-value:

Suppose, during the execution of the A* algorithm, we pop out a node with state S from the priority queue. **Then the cost of any node that can be found via S is no less costly than f(S).**

To go to S the cost is exactly g(S), to go to the goal state from S we need at least h(S) more because h never over-estimate.

- Then **what is the f-value of a goal state?** Simple: it must be exactly the g-value, i.e., the total path cost, since the h-value is 0.
This is by definition on heuristic.
- If we can pop out a goal state G, it means **all unexpanded states must have f-value to be at least f(G)**, so all goals reachable from them have total path cost at least f(G), which is the path cost of G.
So all other path cost no less. G must be optimal.

AI(0270)-2.48

Costs of A* search

How many nodes will get expanded?

- Any node with **f-value less than the path length** will be expanded.
- All the children of these nodes will be pushed into queue.

So how good the algorithm performs depends on the number of nodes having f-value less than the minimum path length.

If the heuristic is better (i.e., larger h), we will **expand less nodes**.

Because a large f means less node will have small f-value.

Can we get better searching algorithms? Yes and no.

No: we **cannot reduce the number of nodes expanded**.

Yes: we **can reduce the memory cost** by forgetting some of the nodes.

If interested: read RBFS and SMA* of the textbook. One can also use IDA* (like IDS), but it is more difficult to have the "next iteration".

AI(0270)-2.49

How to make admissible heuristics

- One question: **how to come up with admissible heuristic?**
- Many problems are difficult only because there are some **restrictions on the actions**.
- If we estimate by **removing some of these restrictions**, we usually get a reasonable admissible heuristic. E.g.,

problem/heuristic	Restriction removed
map geometric distance	must not leave the roads
8-puzzle No. of misplaced pieces	Can only slide pieces
8-puzzle Sum Manhattan dists	Cannot slide through pieces

You will probably need some imaginations to find the restrictions.

AI(0270)-2.50

Combining admissible heuristics

If we get two admissible heuristics, we can determine if **one is better**:

- A heuristic h_1 no worse than another heuristic h_2 if h_1 is **no less than h_2 for all states**.
Be reminded that an over-estimating heuristic is not admissible, so is not better. To be larger, one must be more accurate.

- Then we say h_1 **dominates** h_2 .

So what to do if we get two heuristics h_1 and h_2 with one larger for some nodes and the other larger for the other nodes?

i.e., **both are not better than the other**

Then we can find a heuristic that is better than both h_1 and h_2 by combining them:

The heuristic $\max(h_1, h_2)$ is admissible.

AI(0270)-2.51

Using pattern database

- For the sliding tile problem, there is indeed one even more drastic heuristic. The idea: **remember the cost of some subproblems**. E.g.,

4	?	
?	1	?
?	3	2

⇒

1	2	3
4	?	?
?	?	

We find the minimum number of moves involving 1, 2, 3 and 4 for the above.

- By increasing the number of "wildcard" tiles, we can get less subproblems to remember, in the cost of less accurate heuristic.
- The real 8-puzzle is the **combination** of 2 such subproblems.
- E.g., one subproblem gives a minimum number of moves involving 1, 2, 3 and 4; the other gives that involving 5, 6, 7 and 8. Adding them up gives a heuristic much more accurate than Manhattan distance.

AI(0270)-2.52