

More about repeated state (for last lecture)

Lecture 3

Better search: heuristics

We have seen un-informed searching is in general not very efficient, and can only search to a very limited depth.

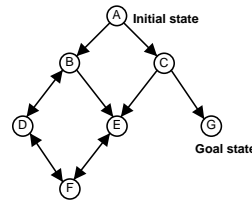
In this section we see methods that **make use of problem specific knowledge** to make search much more efficient.

Reference:

- Textbook sections 4.1 and 4.2.

In the last lecture, we talk about three ways to deal with repeated states. Here we gives one example and see how well the three ways performs given that the search algorithm used is **DFS**.

The SS:



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.

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.**
We will see that this is very wrong: if we know what problem we are solving, we usually have some idea about the structure of the state-space.
- In reality, we usually get **only rather inaccurate information** about the state-space.
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

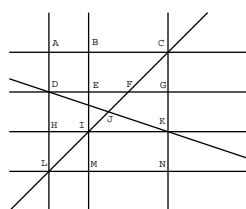
- So we want a **general way** to express **domain knowledge** (i.e., properties of all SS that comes from the same problem) in **searching.**
General way to "express" domain knowledge, not general way to "find" domain knowledge. The latter is impossible: it is called domain knowledge for a reason.
- 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.**
This assumes that the path cost of a path form by some operators is the sum of costs of all operator involved. Luckily, this still covers many problems.
- Any such function is called a **heuristic function**, or a **heuristic.**
This is the usage when we are talking about search algorithms. In most other cases, heuristic means "rule-of-thumb" that improves average case performance of an algorithm.

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.

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—it simply takes no operator at all.
- The function always returns a **non-negative real number.**
In other words, there is no negative operator cost.

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

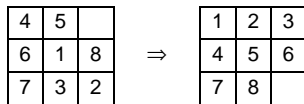
A **special property:** if an estimate **never overestimate the path cost**, it is called **admissible.**

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

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

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)-3.6

Hill climbing

Once we know what is a heuristic, the next matter of affairs is to consider **how to use a heuristic**.

The simplest way to use a heuristic is called "**hill-climbing**".

Also called gradient descent, if you consider the optimal being the bottom of a pit.

- Repeatedly go to the next-state with the least estimate of cost-to-target.

Performance:

- **Memory cost:** constant (need only to store a fixed number of states)
This is really very impressive!
- **Time cost:** no real bound (it can loop, as we will see).
- **Complete? No. Optimal? No.**
It is easy to understand that this algorithm need to remember few things: it is pretty dumb.

AI(0270)-3.7

Code

The code of hill climbing is more or less trivial.

```

State HillClimb(State init_state) {
    State s = init_state;
    while (!GoalState(s)) {
        bool first = true;
        for (State s1 in NextState(s)) // Need some fixup
            if (first || h(s1) < h(s)) { // h is the heuristic
                first = false;
                s = s1;
            }
        }
    }
    return s;
}

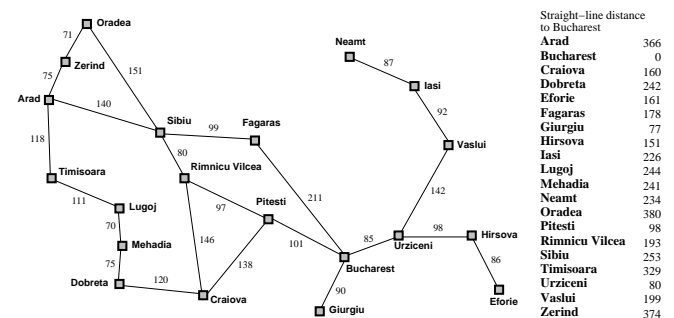
```

From now on, we will call the heuristic function of a state S to be the "**h-value**" of the state, and denote it as $h(S)$.

AI(0270)-3.8

Example

Let's see what will happen when applied to our first example problem:



The heuristic: straight-line (geometric) distance

AI(0270)-3.9

Good, bad and ugly

How good hill-climbing performs in the map of Romania with this heuristic? Let's see:

- A **good** example: From Dobreta to Bucharest
Found path from Dobreta to Craiova to Pitesti to Bucharest. (Optimal)
- A **bad** (or not so good) example: From Sibiu to Bucharest
Found path from Sibiu to Fagaras to Bucharest (Suboptimal, the path via Rimmnicu Vilces and Pitesti is better)
It is very short-sighted. It only cares to get us (seems) closer to the goal, but don't care about the cost to perform the step.
If this doesn't look bad enough, delete the link between Fagaras and Bucharest, and add back a link between Fagaras and Craiova.
- An **ugly** example: From Dobreta to Bucharest, with Dobreta-Craiova road closed.
Oscillate between Mehadia and Dobreta.

AI(0270)-3.10

Adding backtracking: Greedy (Best-first) search

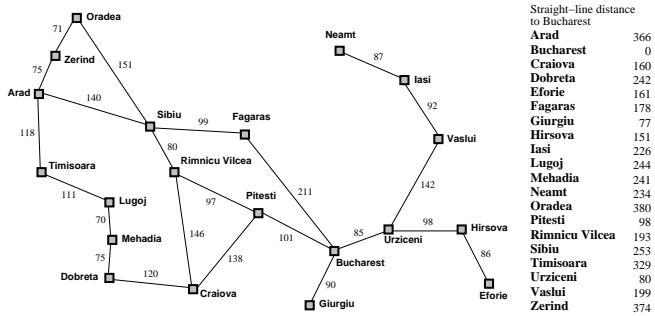
Now we see that **just going to the seemingly best place won't work**. But later, we will see some use the algorithm in other settings.

- Remember that all our previous algorithms doesn't look like hill climbing. **Instead, they memorize the things that have been tried.**
- How about **slightly modifying one of these algorithm**, e.g. BFS, so that it makes use of the heuristic function?
- The intuitive solution: **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**.

AI(0270)-3.11

Example

Let's redo the example and see what we get.



AI(0270):3.12

Example (cont'd)

- From Dobreta to Bucharest: **basically the same optimal result**, except that Mehadia and Rimnicu Vilcea are still in the queue.
- From Sibiu to Bucharest: again, **basically the same sub-optimal result**, except that Arad, Oradea and Rimnicu Vilcea are still in queue.
- From Sibiu to Bucharest with edges between Fagaras and Bucharest modified: still, **same sub-optimal result**.
- From Dobreta to Bucharest with Dobreta-Craiova closed: goto Mehadia, try back to Dobreta: repeated, goto Lugoj, try back to Mehadia: repeated, goto Timisoara, try back Lugoj: repeated, goto Arad, then Sibiu, and then the same as second case.

Bottom-line: it is complete, sub-optimal but usually do well.

AI(0270):3.13

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**.

Backtrack here means to completely ignore a node rather than expanding it.

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.

But greedy search also has its unique problems:

- Time cost:** in trees: size of tree. (In graph: the size of the "expanded" tree (that contains repeated states).) (Same as DFS.)
- Memory cost:** in trees: **can be as large as the tree!**

AI(0270):3.14

A easy fix

- Is there anything we can do about the **stupidity of greedy search** (like the second sub-optimal example)?
- 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)**!
E.g., for the first sub-optimal solution, the cost from Fagaras (with goal Bucharest) involves the distance from Fagaras to Bucharest (h-value), but also the cost from Sibiu to Fagaras.
- There is one very simple thing we can do: **don't forget about the cost we need to go from the initial state!**
- So the fix is easy: to **replace the priority function by the inverse of the sum of g-value and h-value of the state!**
This sum is called the f-value of the state.

AI(0270):3.15

Special case: admissible heuristic and A* search

In general, the algorithm has the same worst case behaviour as greedy: perhaps the heuristic is very bad, leading us to an infinite subtree.

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**.

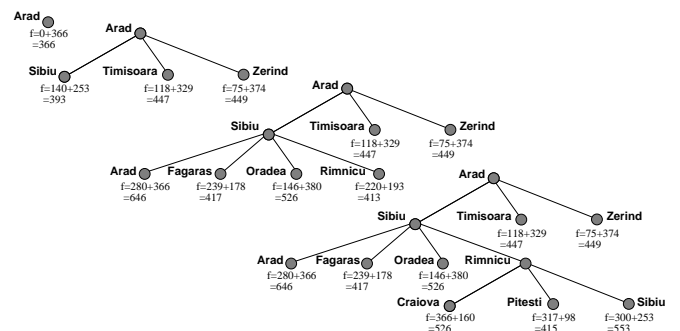
Let's use the Romania map as an example again.

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

Suppose we want to go from Arad to Bucharest...

AI(0270):3.16

Example (cont'd)



AI(0270):3.17

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 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 cost cost no less. G must be optimal.

AI(0270)-3.18

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 IDA* in textbook. The idea is similar to IDFS: doing it depth-first but limiting the search depth (more exactly, max f-value).

AI(0270)-3.19

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 operators**.
- 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)-3.20

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)-3.21

How good are they?

How good is A* search? Let's consider what happens for the 15-puzzle.

- A* search **can't solve the 15-puzzle** for most instances, because the computer would run out of memory.
- IDA* (the memory-preserving variant) can solve the problem using reasonable amount of time and memory.
- But IDA* can't handle the larger variant, 24-puzzle (too much time needed).
- IDA* expands more nodes, but is generally faster because of the memory saved, and because it is mostly depth-first.
A depth-first search allows recursive implementation without a separate queue. But of course, different SS and heuristic performs very differently.

But there is one bigger problem: what for those problem with cost not equal to the sum of operator costs?

AI(0270)-3.22