Lecture 2

Searching in AI

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

We will see most of the techniques to perform searching if we know very little about the problem domain.

References

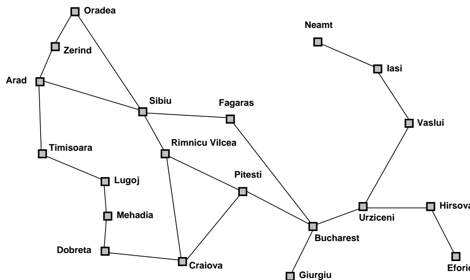
- Textbook section 3.1–3.6.

Background of AI: searching

- Most AI problems are concerned with **searching through many alternatives** to seek a good way to **achieve a goal**.
  Still remember the notions of rational agent and performance measure?
- For many problems, searching is the only thing one needs to do. E.g.:
  - Substitute letters by digits to make SEND+MORE=MONEY correct.
  - Place 8 queens in the chess-board so that none is threatened.
  - For “harder” problems, searching is usually a **subroutine** that can be used to solve the problem.
    - Some such problems: theorem proving, expert systems, etc.
    - *The techniques that are used for searching are quite similar*, so we will learn it once and for all.

What is searching

- If you are in Arad of Romania, and want to go to Pitesti. If you have no knowledge about where are the two places, you would be rather lost.
- But if you get a map, you can looking for a route between them.



What is searching: cont’d

- Once you find a route, you can follow the route to complete your journey. (i.e., **execute** the resulting plan)
- “Searching” is to **find a route** from a starting place to some predefined ending place, by **using local knowledge** about how to reach one place from another.
- But the problem is, **how to perform the searching**?
  This is an especially big problem when the map is large.
- Unlike searching in a search tree, the information is **usually not designed for efficient search**.
- That’s why searching in **AI pose us much more difficult problem** than searching in search trees.
  Recall that searching in AVL tree (or red-black tree) only requires $O(\log n)$ time. Our search will be much much more expensive.

Formulating problems: state space

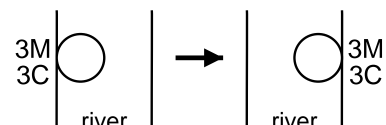
But wait… **where do we want to search within**?
“I don’t sense that there is a map! What map you’re talking about?”

- When we talk about a “map”, what we really mean is a **graph**.
  Not the type of graph that has an X and Y axis, but the type that has nodes and edges. If you’re not familiar with it, just think it as a generalized tree.
- Many problems are formulated as requiring to achieve a certain goal, achieved step by step through **intermediate states**.
- **Each possible state is a node** of the graph we search in (the graph is thus called the “**state-space**”: space of states).
- **“Operators” takes you from state to state**, and form edges.
- The graph is **implicit**, i.e., most likely we don’t want to create the whole graph in memory before we actually perform the search.
  Because the search graph can be very large, and we won’t need it after all.

Example: Missionaries and cannibals

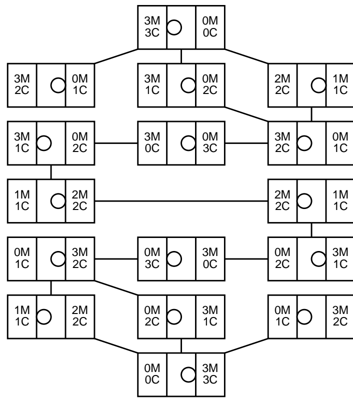
Question: we have **3 missionaries** and **3 cannibals**, all want to cross a river with a **boat**, which can carry **one or two people**.

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**.



- States: include **how many missionaries and cannibals** are in each side, and **where is the boat**.
- Edges: connecting two states that can be reached by a **single boat trip**.

State space



Once you have the state-space graph, the problem is clearly one to search the graph and find a path from one node to another.

As a human being which neurons are trained for searching visually in a small graph, you should find a path right away now.

Actually, it is this easy only because I draw it in a very structured way.

But **what the program should do to perform such a search?**

Representing a state

Now, it comes to **represent a state** in our program. It turns out to be extremely simple: all we need is to have a **struct** that holds all the information.

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

Some of you might be accusing me of wasting so much memory.

The **initial** state is { {3, 0}, {3, 0}, 0 }.

The **final** state is { {0, 3}, {0, 3}, *x* } for any x.

In fact, the operators will only lead to where x=1, but we shouldn't know it beforehand.

The next state function

Now comes the hard part: **how to represent the graph**?

We **won't generate the graph in memory**, in preparation of problems that are much larger.

If you have taken DAA and learnt about adjacency matrix and adjacency list, don't forget about them yet.

Instead, we will represent it by **a function that, given a state, generate all the possible next states.**

Functionally, this is equivalent to an adjacency list.

The real trouble is to **store the states**. We will deal with that problem when we talk about the *search algorithms*. Currently, let's

> Assume that the states need not be stored, and we just need to call a function `f` for each state.

After all, we've generated the state, we just need to find a way to use it.

Writing the next state function

It should be more or less trivial: just consider all the possible changes to the states.

```
void next_state(mc_state s) {
    int p = s.boat_pos;
    s.boat_pos = 1 – s.boat_pos;
    --s.m[p];
    ++s.m[1–p];
    if (valid(s)) f(s);
    --s.m[p];
    ++s.m[1–p];
    if (valid(s)) f(s);
    ++s.m[p];
    --s.c[p];
    --s.m[1–p];
    ++s.c[1–p];
    if (valid(s)) f(s);
    ... // 2 more possibilities
}
```

```
// Out-Of-Bound
bool oob(int num) {
    return num < 0 || num > 3;
}

bool valid(const mc_state &s) {
    if (oob(s.m[0])) return false;
    if (oob(s.m[1])) return false;
    if (oob(s.c[0])) return false;
    if (oob(s.c[1])) return false;
    if (s.m[0]≠ 0 && s.m[0] < s.c[0])
        return false;
    if (s.m[1]≠ 0 && s.m[1] < s.c[1])
        return false;
    return true;
}
```


Searching: formal definition

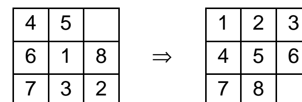
Now it is instructive to see **how this generalizes** to so many problem solving tasks. Each problem consists of:

- **States**. Represention: a structure.
- **Operators** to travel among states. Representation: a next-state function, generating all neighbouring states of a state.
- **Initial** state. A state which the search is started.
- **Goal** states. A set of states which are said to achieve the target. Representation: Either a state, or a function returning whether goal is achieved.
- **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**.

  Path cost is usually the sum of cost of operators used to reach the goal.

Example 2: 8-Puzzle

Example problem:



- **States**: configuration (representation: string of 9 characters).
  Or any other reasonable representation, e.g., 2-D array.
- **Operators**: push-left, push-right, push-up, push-down.
  Number of operators 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 operators. (Less is better.)
- **Initial and goal state**: defined by problem instance.
- **Path cost**: the cost of each operator is 1, path cost is sum of it.

Example 3: Arithematic puzzle

Substitute letters by distinct digits to make the following calculation correct.

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

- **States**: a partial assignment of letters to digits
- **Operators**: adding an assignment. (Many!)
- **Initial state**: no assignment is made.
- **Goal**: all letters are assigned a digit, and the formula is correct.
- **Path cost**: 0 (good as long as you can find an assignment).

Example 4: Tic-tac-toe

- **State**: a placements of circles and crosses on the board.
- **Operators**: add another circle at a particular position of the board, and add another cross at a particular position of the board.
- **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 3.

Our main point here: **many** (I can't even say most…) problems can be **formulated and solved as a state-space search problem**.

Search algorithms

Now it finally comes to solving 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, arithematic puzzle, 8-queen, etc), we can afford to search it exhaustively.
  Search space size of all the 3 problems are around 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 the heuristics that we will explain in the next lecture.
- 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.

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.
- **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.

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).

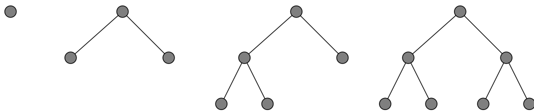
Breadth-first search

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 states.
- At the beginning, the queue simply contains the **initial** state.
- In each iteration:
  - One state is **de-queued**.
  - All the **valid next-states are generated**.
  - If any is a **goal state**, the search terminates.
  - Otherwise, they are **en-queued** into the queue.

Example

One example, where branch factor is 2:



We usually call the operation to *dequeue a state*, find all *next states* and *enqueue them back* as **expanding a state**.

So the second picture expand the root node of the tree, and the third and fourth expand the 2 first-level children.

Why we get tree instead of graph? Let's simplify it a bit, we'll come back to deal with cycles soon.

Code

The code is rather simple:

```
State BFS(State init_state) {
    queue<State> q;
    q.push(init_state);
    while (!q.empty()) {
        State s = q.front();
        q.pop();
        for (State s1 in NextState(s)) { // Need some fixup
            if (GoalState(s1))
                return s1;
            else
                q.push(s1);
        }
    }
}
```


How good is it?

Okay, how good is breadth-first search? In simple words, **not very good**.

- **Complete. Optimal** if each operator costs the same.
- 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 $1 + b + \ldots + b^d$. This is the **time cost**.
- **Memory cost**: store b^d states.
  Only the last layer is still in queue, all the others have been dequeued.

What if operator costs differently?

- So what should we do if we want optimality but **operator costs are not all the same?**
- **We just need to modify BFS a bit**: how we store the queue.
- All we need to do is **to use a priority queue**, with priority defined to be inverse of the path cost.
- We will also have to make sure that the function **return only when a goal state is about to expand.**
  Not when a goal state get expanded. Note the difference: a state get expanded when it is generated, a state is about to expand when it is de-queued from the queue.
- Memory and CPU costs now depend on the number of nodes that **costs less than the least cost goal node**.
  And the branching factor.

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 programs 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).

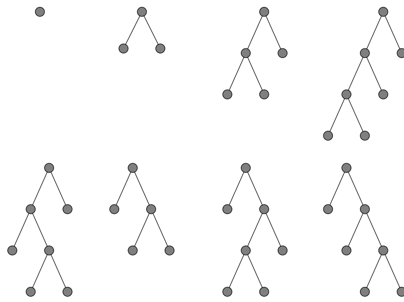
Depth-first search

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**: very simple. Just replace the queue in Breadth-first search by a stack. Done.
  Or you can use recursion.
- **How good is it?** No very good, unfortunately. The primary problem is that it might get stuck in an unrelated tree.
  - **Complete** only if the **tree is finite**. **Not optimal**.
  - **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!)

Example

If we search in a uniform tree with branching factor = 2 and depth = 3, we have this search sequence:



Note that nodes disappears after the visited completely: the memory is freed by then.

Dealing with loops

- As we have seen, **the SS of most AI problems are not trees**.
- **BFS**: **some nodes will be visited and expanded twice**, ending up with more nodes than it requires. But the algorithm will **still terminate**.
- **DFS**: one would probably **loop in a cycle**.
  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.
  - **Remember all expanded states**, and ignore them subsequently.
    Usually, this requires too much memory.

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.
- Is it possible to **combine the goods** of both algorithms? It turns out that it is possible.
- The idea is simple: if we **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.
- The algorithm is called **Iterative DFS (IDFS)**. It simply repeatedly invoke DFS, with a depth limit increasing from 1 until search succeed.
  Isn't it wasteful to have to recompute everything every time we do DFS?

Code

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

```
State IDFS(State init_state) {
    for (int i = 0; ; ++i) {
        State result = DFS_with_depth_limit(init_state, i)
        if (result ≠ none)
            return result;
    }
}
```

How to implement DFS_with_depth_limit? Just the same as DFS, except that **the state would remember the search depth** as well.
In other words, the struct of the state will contain one more int, which denotes the depth. It might have to contain a whole path of states as well to deal with cycles.

We just **ignore any state with depth exceeding the limit**.

How good is it?

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

The basic reason is that **the cost to search the last level usually dominates the cost cost**.
That is, searching in the previous levels are essentially free.

IDFS is

- **Complete**.
- **Optimal** if the cost per link visit is uniform.
- **Time cost**: $O(b^d)$
- **Memory cost**: $O(bd)$
  So being a little bit less aggressive in minimizing CPU usage buys us a lot of memory.

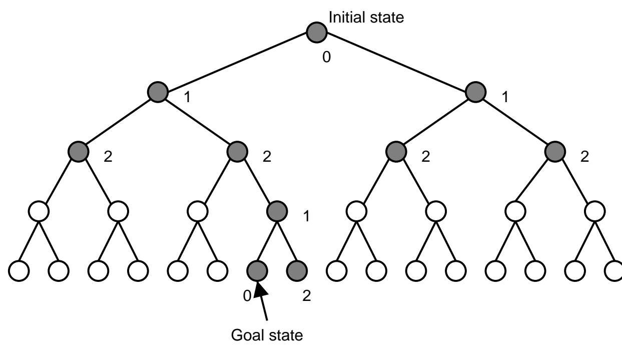
Bidirectional search

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

- 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 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!
  In other words, not really that much better.

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 are **not really that far** from the goal state.
- We have **a lot of memory** to play with.
- (Most important) we have an explicit goal state.
  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.

Also, **the space requirement is actually a bit more** than storing two smaller trees because one has to keep a hash table for matching.

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**.

Conclusion

- We have seen some **very general search strategies** here that does not require us to know anything about the problem domain.
- In general, the branching factor does not allow us to search for more than a few dozen steps.
- Only that distance can be handled using such search.
- **Are these search useless?** Not quite. They are the basis of more advanced searches, and usually they are the only option that we can choose.
  Even the most sophisticated algorithm uses makes some use of DFS or BFS.
- But if we **know better** about the problem, we would choose another strategy for searching.