Search Algorithms

Matteo Fasiolo

What is Search?

Search algorithms find object(s) from **a candidate set** according to a **searching criterion**.

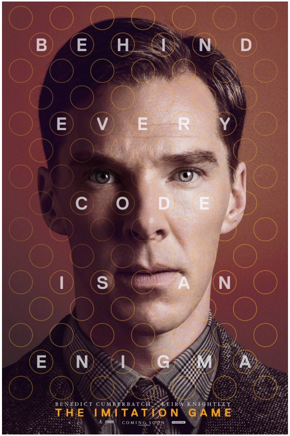
Many CS problems are search problems:

- Find the index of the maximum (smallest) element in an array.
- Find the index of a specific element in a sorted array.
- Find x that maximizes a function $f(x)$.
- Find the best move in a chess game.
- Find the key to a password encryption.

Cryptography

Cryptography is a classic search problem.

To decrypt messages, you need to find “keys” from a huge candidate set: The mathematician (Alan Turing) who tried to solve this problem invented modern computers.



Machine Learning

Many machine learning problems are search problems:

Find a model from a **model family** that minimizes/maximizes objective functions:

- A good weather prediction model should accurately predict temperatures with minimum error when comparing to the actual weather data.
- A good advertisement model should recommend user ads that leads to maximum click rates.
- Commonly used model family (such as neural networks) usually a huge search space.

Search Problem

More formally,

$$\{i^*\} = \operatorname*{argmin}_{i \in \mathcal{F}} f(i)$$

where $\mathcal{F}$ defines a search space and f defines the search criterion.

For example, finding the smallest element in a length n array:

$$\{i^*\} = \operatorname*{argmin}_{i \in \{1,\dots,n\}} s_i,$$

where s_i is the i-th element in the array s.

Finding the index of the element closest to 4 in a length n array:

$$\{i^*\} = \operatorname*{argmin}_{i \in \{1,\dots,n\}} |s_i - 4|,$$

When the search space is discrete, and is not large, we can enumerate the entire $\mathcal{F}$ to find the best fit(s).

- Like what we have done in `find_max_idx` function.

However, what if the search space is infinite?

$$\{x^*\} = \arg_{x \in \mathbb{R}} \max f(x),$$

Do we search the entire real domain?

Local Search Algorithms

Local search algorithms is the name of **a set of search algorithms**.

1. Local search algorithm starts from a candidate solution a problem.
2. It searches for a better solution that is close to the current solution.
3. If it find a better solution, it searches for an even better solution in its neighbourhood.
4. Keep iterating until a certain stopping criterion is satisfied.

For local algorithms to work, the local problem must provide **enough information** to the original problem.

Hill Climbing

Consider the problem of finding the maximiser of a function:

$$\{x^*\} = \underset{x \in \mathbb{R}}{\operatorname{argmax}} f(x),$$

How do we split this problem into meaningful local subproblems?

We can start at an arbitary location x_0 and solve the subproblem:

$$\{x_1^*\} = \underset{x \in [x_0 - \epsilon, x_0 + \epsilon]}{\text{argmax}} f(x),$$

where ϵ is a fixed value.

The next step, we solve

$$\{x_2^*\} = \underset{x \in [x_1 - \epsilon, x_1 + \epsilon]}{\text{argmax}} f(x)$$

and so on...

- After finding the maximizer of the previous local problem, we restart search for the maximum centered around the previous maximizer.
- Hence the name "Hill Climbing".

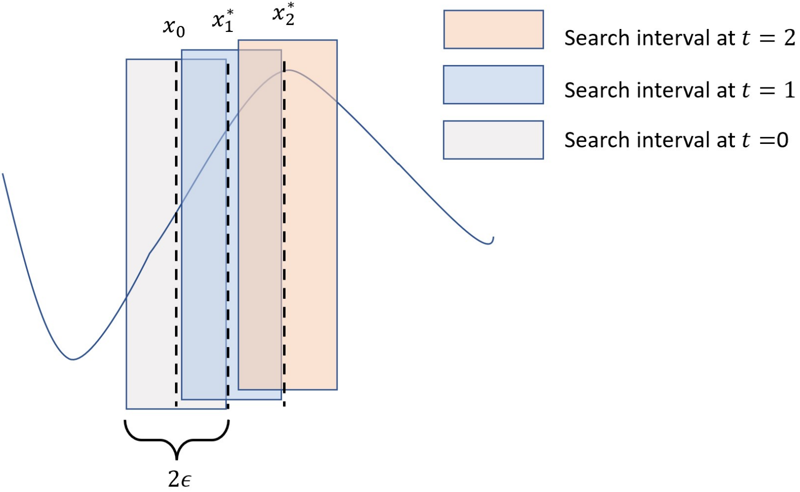
The algorithm stops when successive steps finds identical optimal solutions, i.e. $x^*_{t+1} = x^*_t$.

For each sub-problem,

$$\{x^*_{t+1}\} = \underset{x \in [x_t - \epsilon, x_t + \epsilon]}{\text{argmax}} \; f(x),$$

we can find an approximate solution via grid search:

1. Split $[x^*_t - \epsilon, x^*_t + \epsilon]$ into discrete "grid points".
   - e.g. $[-1, 1]$ can be discretized as $[-1, -.9 ... .9, 1]$.
2. Search for the grid point that gives the smallest function value.



x_0
x_1^*
x_2^*
2ϵ
Search interval at $t = 2$
Search interval at $t = 1$
Search interval at $t = 0$

The Subproblem

```
// This function returns xmax that maximizes f(x)
// between x-epsilon and x+epsilon
// complete the code yourself
double subproblem(double xt, double epsilon){
    double x = xt - epsilon;
    double fmax = -100; // Sloppy, better: -DBL_MAX
    double xmax = x;
    while(x <= xt + epsilon){
        if(_____){
          //fill out the if statement
        }
        x += epsilon/100;
    }
    return xmax;
}
```

Write the main function

```
int main(){
  double epsilon = .5;
  double x0 = -4; //initial point

  double xt = x0;
  double xt_1 = 100; // xt-1

  while( fabs(xt - xt_1) >= 1e-5){//fabs absolute value
    xt_1 = xt;
    // _________ // fill out the blank
    printf("f(%.4f) = %.4f\n", xt, f(xt));
  }

  printf("Maximum is at %f with value %f\n", xt, f(xt));
}
```

Output

```
f(-3.5000) = -0.3430
f(-3.0000) = -0.2658
f(-2.5000) = -0.1380
f(-2.0000) = 0.0361
f(-1.5000) = 0.2489
f(-1.0000) = 0.4895
f(-0.5000) = 0.7448
f(-0.0050) = 0.9975
f(0.4900) = 1.2351
f(0.9850) = 1.4427
f(1.4850) = 1.6080
f(1.9850) = 1.7168
f(2.4850) = 1.7597
f(2.5400) = 1.7602
f(2.5400) = 1.7602
Maximum is at 2.540000 with value 1.760173
```

Questions

1. Is this algorithm guaranteed to find the maximum of $f(x)$?
2. How will epsilon affect the performance of the algorithm?

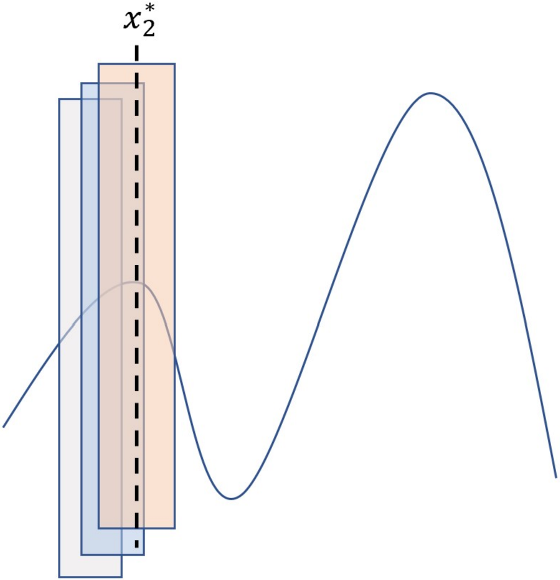
Local optimum

No, the algorithm is **not** guaranteed to find the maximum.

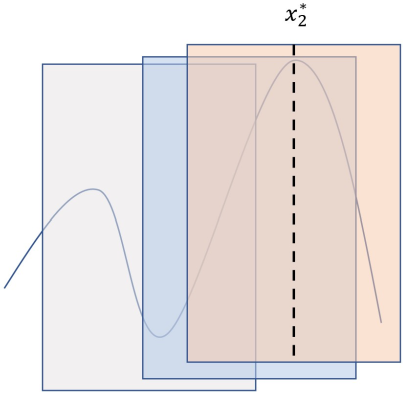
Solving subproblems successively may not lead to the global optimal solution.

The subproblems may not provide necessary information about **the global structure** of $f(x)$ to reach the global optimal solution.

It may only reach a solution that is **locally optimal**, which means the solution is optimal only within **the reach of our search spaces**.



Algorithm stuck at x_2^*!



With a bigger ϵ, algorithm is **not** stuck at local optimal!

However, bigger ϵ means solving the subproblem is more time-consuming (more grid points).

We have to make the trade off between a better solution and computational time.

TicTacToe

In some cases, the search space $\mathscr{F}$ is finite and well-defined, but the searching criteria is not.

Consider the game TicTacToe:

```
X O *
* O *
X X *
```

where $*$ is an empty spot.

Players X and O try to connect their pieces as a straight line, which can be a column, a row, or a diagonal line.

```
X O *
* O O
X X X // X wins!
```

Given a specific game, what is the optimal move?

```
X O *
* * *
* * *
```

The "moves" are finite and well-defined (the empty spots on the board).

However, how do we define "the optimal move"?

Sure, a good move leads to winning the game, but how to specify the optimality in an algorithmic manner?

A Greedy Algorithm

We can use a greedy approach.

Similarly to local search, the problem (the game) is divided into subproblems (turns).

We make the decision that seems optimal for the current subproblem.

That is, we make a decision considering the outcome of the current turn, not of the whole game.

Hence, at each turn we are greedy (and short-sighted).

Note, hill-climbing can be considered a local greedy search algorithm.

So, instead of aiming at “winning the game”, we focus on the **next turn only**.

Let the game board be a 3 by 3 matrix.

Recall the game is trying to connect pieces into a straight line.

For a good move i, j,

- It should maximize the number of our own pieces on i-th row, j-th column and diagonal lines (if i, j is on the diagonal).
- It should avoid opponent pieces along these lines, which would prevent us from forming a line.

At step t, the optimal move is determined by

$$\{(i^*_{t+1}, j^*_{t+1})\} = \operatorname*{argmax}_{(i,j)\in\mathcal{F}_t} f(i,j),$$

$\mathcal{F}_t$ is the set of feasible moves.

$f(i,j)$ scans i-th row, j-th column and diagonal lines (if i, j is on the diagonal or anti-diagonal).

The function value is the number of self-pieces on these lines **minus** the number of opponent-pieces.

Suppose M is the game board. Let $\sum_k = \sum_{k=1}^3$ and define

$$
\begin{aligned}
f(i,j) := &\sum_k \mathbb{1}(M_{i,k} = \text{self}) - \sum_k \mathbb{1}(M_{i,k} = \text{opp}) \\
&+ \sum_k \mathbb{1}(M_{k,j} = \text{self}) - \sum_k \mathbb{1}(M_{k,j} = \text{opp}) \\
&+ g(i,j) + h(i,j).
\end{aligned}
$$

Count along main diagonal:

$$
g := \begin{cases} \sum_k \mathbb{1}(M_{k,k} = \text{self}) - \sum_k \mathbb{1}(M_{k,k} = \text{opp}), & i = j \\ 0, & i \neq j \end{cases}
$$

Count along main anti-diagonal:

$$
h := \begin{cases} \sum_k \mathbb{1}(M_{k,4-k} = \text{self}) - \sum_k \mathbb{1}(M_{k,4-k} = \text{opp}), & i + j = 4 \\ 0, & i + j \neq 4 \end{cases}
$$

```
class tictactoe{
 matrix board; // the game board
 int isplayable(int i, int j) {// is i,j playable?
  if (board.get_elem(i, j) == '*') {
   return 1;
  }
  return 0;
 }

 public:
    tictactoe(): board(3,3){// how to initialize field
       //TODO: initialize the board with *
    }
    void play(int i, int j, char player) {
        if(isplayable(i, j)) {
           // fill out the blanks
           board.set_elem(__, __, __);
        }
    }
};
```

Implement the following private functions

```
int pieces_at_row(int i, char player){
  //Count player's pieces at i-th row
  int count = 0;
  for(int j = 0; j < 3; j++){
      if(board.get_elem(i, j) == player) count++;
  }
  return count;
}
int pieces_at_col(int j, char player){
  //TODO: Count player's pieces at j-th column
}

int pieces_at_diag(char player){
  //TODO: Count player's pieces at diagonal line
}
int pieces_at_anti_diag(char player){
  //TODO: Count player's pieces at anti-diagonal line
}
```

```
int evaluate(int i, int j, char player){ //compute f(i,j)
 char opponent;
 if (player == 'X') opponent = 'O'; else opponent = 'X';

 int f_ij = 0;
 f_ij += pieces_at_row(i, player);
 f_ij -= pieces_at_row(i, opponent);
 f_ij += pieces_at_col(j, player);
 f_ij -= pieces_at_col(j, opponent);

 if(i == j){
  f_ij += pieces_at_diag(player);
  f_ij -= pieces_at_diag(opponent);
 }
 if(i + j == 4){
  f_ij += pieces_at_anti_diag(player);
  f_ij -= pieces_at_anti_diag(opponent);
 }
 return f_ij;
}
```

Make the Move!

Write the following public functions:

```
void play(char player){
  // TODO: It play the next move for the "player",
  // play the move (i,j) is the maximizer of f(i,j)

}
```

Note that this is in addition to:

```
void play(int i, int j, char player) {
      if(isplayable(i, j)) {
         // fill out the blanks
         board.set_elem(__, __, __);
     }
}
```

Let it Play!

```
int main()
{
    tictactoe game;
    game.play(0, 0, 'X');
    game.play(1, 0, 'O');
    game.play(0, 2, 'X');
    game.play(1, 2, 'O');
    game.print();

    printf("AI plays... \n");
    game.play('X');
    game.print();
}
```

- You may need to implement your own `print` function to print out the game.

Output

```
X * X
O * O
* * *

AI plays...
X X X
O * O
* * *
```

It works!

However, is our strategy optimal?

A Suboptimal Move

```
X O *
X X O
* * *

AI plays...
X O O
X X O
* * *
```

You can design a more complex subproblem anticipating your opponent's counter moves, but it would increase computational complexity too.

We have to compromise between computational time and optimality of our solution.

Conclusion

Search Problem finds objects(s) from a search space given a searching criterion.

Local/greedy algorithms:

- Hill Climbing
- TicTacToe

Both of them can find locally optimal solutions, but neither of them is guaranteed to find the global optimum.

Greedy algorithm is myopic, which means it only focuses on optimizing the subproblem, which may NOT lead to **the global optimal solution**.

Homework 1

Start from `lab_hill_climbing_template.cpp`.

1. Complete the code for the hill-climbing algorithm.
2. Test it on the objective function provided in the code.
3. Experiment with other values of `epsilon` and with other objective functions.

Homework 2

Start from `lab_ttt_template.cpp`.

Note that we have included the `matrix` class from a previous lab.

1. Complete the `tictactoe` class, in particular the public methods:
   - `tictactoe()` (the constructor)
   - `void play(int i, int j, char player)`
   - add a `print()` method
2. Test whether these methods work in `main`.

Homework 2 (submit)

3. Complete the private methods:
   - `pieces_at_col`
   - `pieces_at_diag`
   - `pieces_at_anti_diag`

To test them you might want to make them temporarily public.

4. Complete the public `void play(char player)` method.
5. Test whether it work by initialising from, e.g.:

```
X O *
* * *
* * *
```

and then play `X` and `O` in turn, until one wins.