Search Algorithms

Song Liu (song.liu@bristol.ac.uk)

GA 18, Fry Building,

Microsoft Teams (search "song liu").

What is Search?

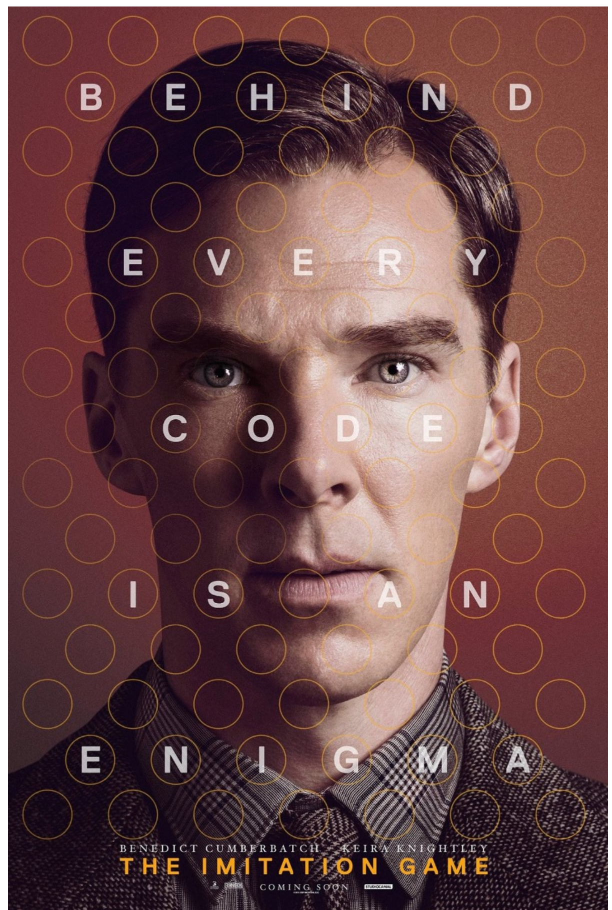
- Search algorithm finds object(s) from **a candidate set** according to a **searching criterion**.

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.

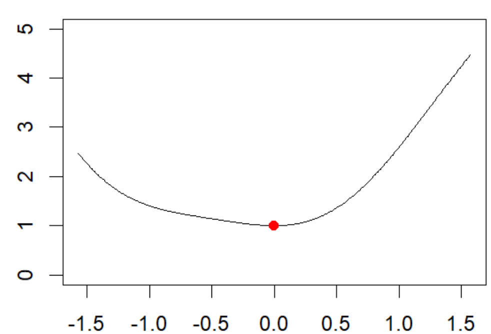
Cryptography

- 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,
$\{i^*\} = \arg_{i \in \mathcal{F}} \min f(i)$

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

For example, finding the smallest element in a length n array:
$\{i^*\} = \arg_{i \in \{1, ..., n\}} \min s_i,$
where s_i is the i-th element in the array s.

finding the index of the element equals 4 in a length n array:
$\{i^*\} = \arg_{i \in \{1, ..., n\}} \min |s_i - 4|,$

Search Problem

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_min_idx` function.

However, what if the search space is infinite?
$\{x^*\} = \arg_{x\in\mathbb{R}} \min f(x)$,
Do we search the entire real domain?

Greedy Algorithm

Greedy algorithm is the name of **a set of search algorithms**.

1. Greedy algorithm starts from a smaller, but more manageable **subproblem**.
2. Greedy algorithm finds the best possible solution (hence the name greedy) for the current subproblem.
3. From the solution, it **revises** the subproblem and solves a new subproblem.
4. Until a certain stopping criterion is satisfied.

For greedy algorithm to work, the subproblem must provide **enough information** to the original problem.

Hill Climbing:

Consider the problem of finding the minimizer of a function:

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

How do we split this problem into meaningful subproblems?

Hill Climbing:

We can start at an arbitary location x_0 and solve for the following problem:

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

where ϵ is a fixed value.

The next step, we solve

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

and so on...

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

Hill Climbing:

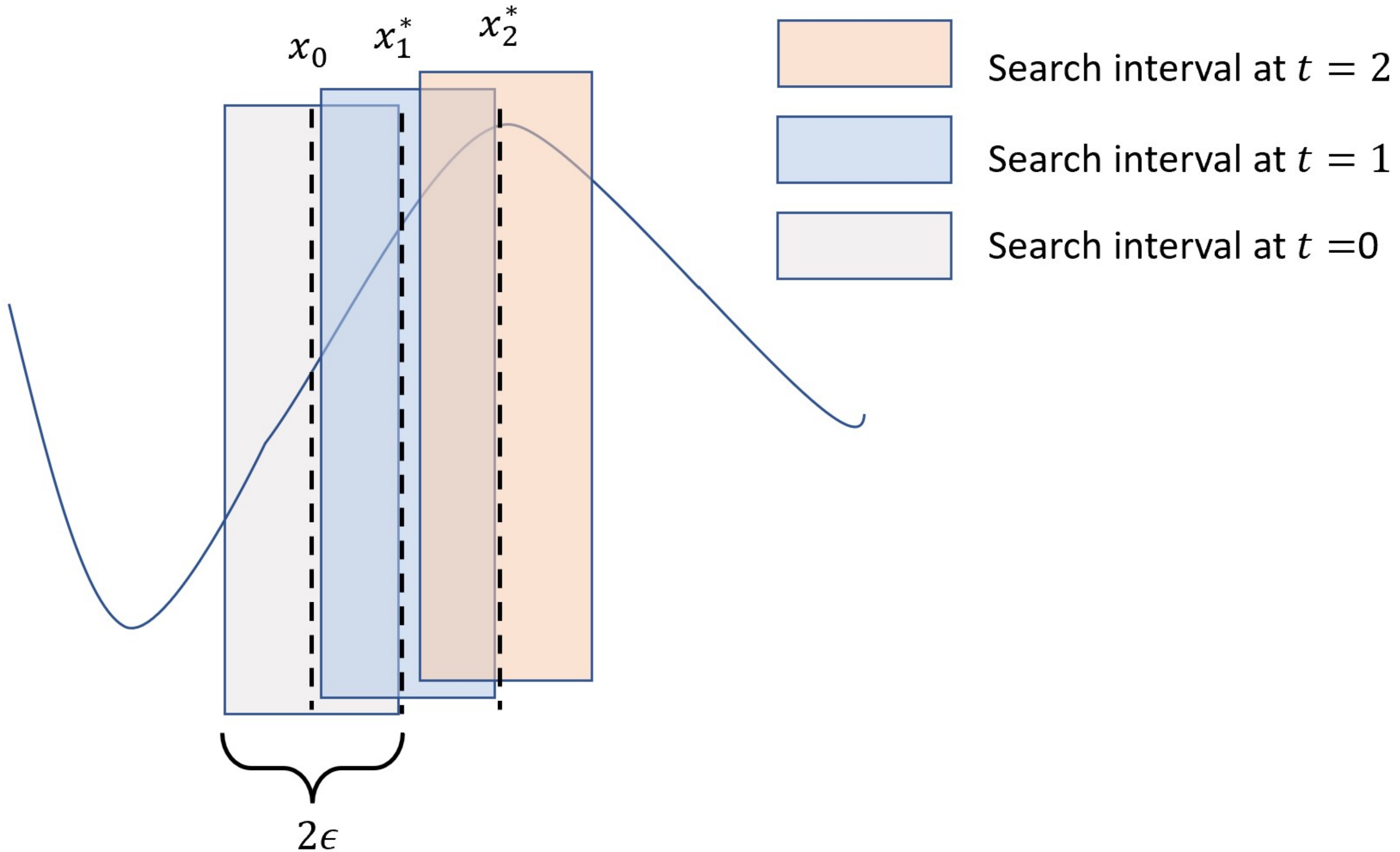
The algorithm stops when successive steps finds identical optimal solutions, i.e. $x^*_{t+1} = x^*_t$.

For each sub-problem,
$\{x^*_{t+1}\} = \arg_{x \in [x^*_t - \epsilon, x^*_t + \epsilon]} \max f(x)$,

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 $[-1, -.9 \ldots .9, 1]$.
2. Search for the grid point that gives the smallest function value.

Hill Climbing:



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;
    double xmax = x;
    while(x <= xt + epsilon){
        if(_____){
          //fill out the if statement
        }
        x += .1;
    }
    return xmax;
}
```

Write the `main` function

```c
void main(){
    double epsilon = .5;
    double x0 = 1;

    double xt = x0;
    double xt_1 = 100;

    while( fabs(xt - xt_1) >= 1e-5){
        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

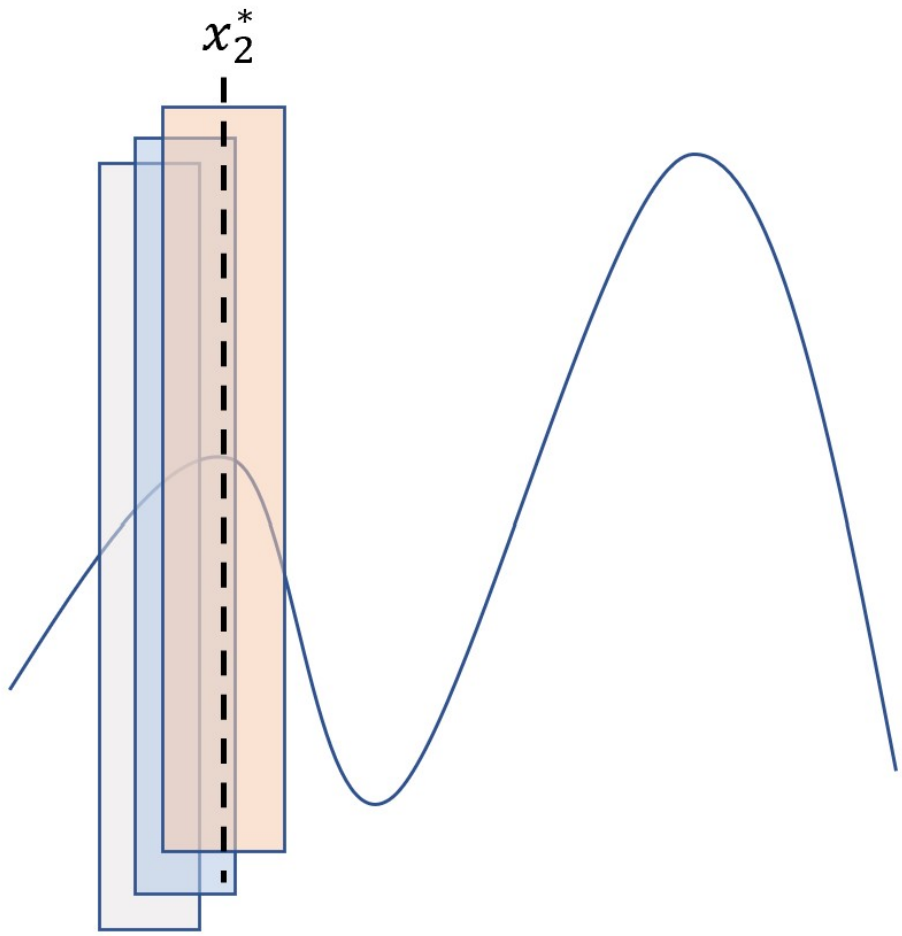
Is this algorithm guaranteed to find the maximum of $f(x)$?

How will epsilon affect the performance of the algorithm?

Locally Optimum

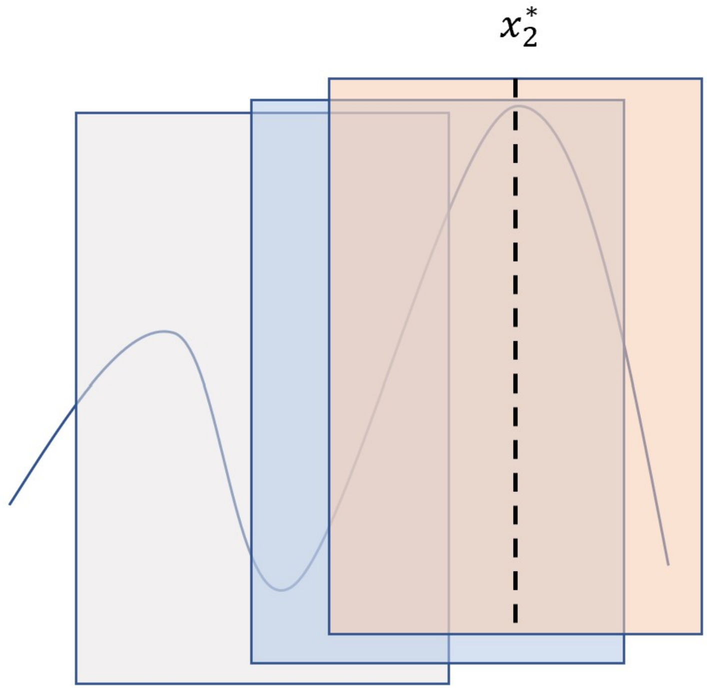
- No, the algorithm is **not** guaranteed to find the maximum.
- Solving subproblems successively may not lead to the global optimal solution.
  - The subproblems may not provide necessary information about **the global structure** of $f(x)$ to reach the global optimal solution.
  - It may only reach a solution that is **locally optimal**, which menas the solution is optimal only within **the reach of our search spaces**.

Local Optimum



Algorithm stuck at x_2^*!

Local Optimum



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

- However, bigger ϵ means solving the subproblem is more time-consuming (more gird points).
- We have to make the trade off between a better solution and computational time.

TicTacToe

- In some cases, the search space $\mathcal{F}$ 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!
  ```

TicTacToe

- Given a specific game, what is the optimal move?

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

- 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?

TicTacToe

- We can use the greedy algorithm.
- Instead of targeting on "winning the game" in the long run, we focus on the **next step 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.

Greedy Algorithm

At step t, the optimal move is determined by
$\{(i_{t+1}^*, j_{t+1}^*)\} = \arg_{(i,j)\in\mathcal{F}_t} \min f(i,j)$,

- $\mathcal{F}_t$ 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.

Greedy Algorithm

Suppose M is the game board. We can define

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

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

$$
h := \begin{cases} \sum_k \mathbb{1}\left(M_{k,2-k} = \mathrm{self}\right) - \sum_k \mathbb{1}\left(M_{k,2-k} = \mathrm{opp}\right), & i = 2 - j \\ 0, & i \neq 2 - j \end{cases}
$$

The Game Class

```
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 you initialize a 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(__, __, __);
        }
    }
};
```

- The `tictactoe` "has a" board.

The Helper Functions

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
}
```

Evaluate Moves

```
int evaluate(int i, int j, char player){
    char opponent = (player == 'X') ? 'O' : 'X';
    // evaluate move i,j
    // the higher the score is, the better the move is
    int score = 0;

    score += pieces_at_row(i, player);
    score -= pieces_at_row(i, opponent);
    score += pieces_at_col(j, player);
    score -= pieces_at_col(j, opponent);

    if(i == j){
        score += pieces_at_diag(player);
        score -= pieces_at_diag(opponent);
    }
    if(i + j == 2){
        score += pieces_at_anti_diag(player);
        score -= pieces_at_anti_diag(opponent);
    }

    return score;
}
```

Make the Move!

Write the following public function:

```
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)

}
```

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
* * *
```

We win!

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.

A Min-Max Search

You can think about an optimal move:

- If there is a chance to win the game in the next round, take it.
- If there is no game-winning moves, take the move that minimizes our opponent chance to win the game.
- Our opponent will use **the same strategy** against us.

This is called **min-max** algorithm.

Conclusion

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