Lecture 5
Local Search

Alice Gao

November 2, 2021

Contents

1 Learning Goals **3**

2 Introduction **3**
2.1 Properties of Local Search . . . 3
2.2 Components of a local search problem . . . 4
2.3 Formulating 4-Queens as a Local Search Problem . . . 5

3 Greedy Descent **8**
3.1 Properties of Greedy Descent . . . 8
3.2 Definitions of Local and Global Optimums . . . 8
3.3 Practice Questions for Local and Global Optimums . . . 9
3.4 Escaping Flat Local Optimums . . . 11
3.5 Performance of Greedy Descent with Sideway Moves . . . 12

4 Greedy Descent with Random Moves **13**
4.1 Two Types of Random Moves . . . 13
4.2 Greedy Descent with Random Restarts . . . 14

5 Simulated Annealing **16**
5.1 Brief Recap . . . 16
5.2 Introducing Simulated Annealing . . . 16
5.3 The Probability of MOving to a Worse Neighbour . . . 16
5.4 Algorithm . . . 18
5.5 Annealing Schedule . . . 19
5.6 An Analogy . . . 19

6 Population-Based Algorithms **21**
6.1 Beam Search . . . 21
6.2 Stochastic Beam Search . . . 22
6.3 The Genetic Algorithm . . . 23

6.4 A Fun Genetic Algorithm Car Simulator 23
6.5 Greedy Descent v.s. The Genetic Algorithm 23

7 Practice Problems **25**

1 Learning Goals

By the end of the lecture, you should be able to

- Describe the advantages of local search over other search algorithms.
- Formulate a real-world problem as a local search problem.
- Verify whether a state is a local/global optimum.
- Describe strategies for escaping local optima.
- Trace the execution of Greedy Descent, Greedy Descent with random restarts, simulated annealing, and genetic algorithms.
- Compare and contrast the properties of local search algorithms.

2 Introduction

First, why do we want to use local search algorithms? Let's think about some issues of the search algorithms we have discussed so far.

THe first issue is that, the algorithm tries to explore the entire search space systematically. The algorithm visits the states in a certain order and it may visit a lot of the states before finding a goal. There are some obvious problems with this behaviour. If the search space is big, systematic exploration will take a long time. If the search space is infinite, we cannot hope to visit all of states. Can we find another search strategy, which allows us to find a solution quickly, without attempting to visit all of the nodes systematically?

The second issue is that the search algorithm remembers and returns a path from the initial state to the goal state. Is this necessary for every search problem? For sliding puzzles, our goal is to find a path from one state to another, so it's necessary to remember the path. For a CSP, such as 4-queens, all we care about is finding a state that satisfies all the constraints. We don't care about the process of reaching that state — the order in which I place the 4 queens on the board or moving them around really doesn't matter. Only the final board matters. Local search is designed to address both issues. First, local search does not attempt to explore the search space systematically. Second, local search only remembers the current state and does not keep track of a path to the goal node. By giving up these properties, what do we gain instead?

2.1 Properties of Local Search

Let's look at some properties of local search algorithms.

First, local search algorithms give up on exploring the search space systematically.

The advantage of this choice is that, instead of attempting to visit all of the states, local search uses strategies to find reasonably good states quickly on average. This is often

good enough in practice especially when we are solving a challenging problem under time constraint.

The downside of this design choice is that local search is not guaranteed to find a solution even if a solution exists. Furthermore, it cannot be used to prove that no solution exists. Thus, we often use local search when we know that a solution exists for sure or a solution very likely exists.

The second design choice is that, local search does not remember a path to the goal state. A nice consequence of this is that local search algorithms require very little memory since they only need to remember the current state.

Last but not least, local search can solve different types of problems, in particular, pure optimization problems. For a pure optimization problem, we have an objective function, but we do not know the best achievable objective value. For instance, suppose that we want to assign radio spectrums to radio and TV stations to minimize interference, but we do not know the minimum amount of interference that can be achieved.

2.2 Components of a local search problem

Next, let me get into the mechanics of a local search algorithm.

To apply local search, we will use a complete-state formulation instead of an incremental-state formulation.

Recall our search problem formulation for 4-queens. We start with an empty board and build up the state by adding one queen at a time. This is called an incremental formulation.

For local search, we start with a complete state where all the variables have assigned values. At each step, we will modify the state based on our neighbour relations, trying to change it to a goal state. For 4-queens, we would start with a board with 4 queens on it, and move one queen at a time until the 4 queens do not violate any constraint.

Let's look at the components of a local search problem.

Example: A local search problem consists of:

- A state: a complete assignment to *all* of the variables.
- A neighbour relation: which states do I explore next?
- A cost function: how good is each state?

First, we need to define a state. The state is a complete state where all the variables have assigned values.

Next, we need to define a neighbour relation. This is analogous to the successor function. A neighbour relation tells us how to modify the current state to generate a new state. Given the current state, which states can I explore next?

Finally, we have a cost function. The cost function takes a state and evaluates the quality of the state with respect to our objective function. Our goal is to minimize the cost of the current state.

2.3 Formulating 4-Queens as a Local Search Problem

Let's take the 4-queens problem and formulate it as a local search problem.

Example: Formulating the 4-queens problem as a local search problem.

First, I will define the state. The state contains variables. The definition of the variables is the same as before. We assume that there is one queen per column and will keep track of the row position of each queen. The domains are the same as before. I will skip the constraints, and you will see why later.

Solution:

Variables: x_0, x_1, x_2, x_3 where x_i is the row position of the queen in column i, where $i \in \{0, 1, 2, 3\}$. Assume that there is one queen per column.

Domains: $x_i \in \{0, 1, 2, 3\}$ $\forall i$.

Next, let me define the initial state and the goal state. The initial state has four queens on the board. There's one queen in each column, but each queen can be in a random row position.

For the goal state, we have four queens on the board and they satisfy all the constraints, which means no pair of queens are attacking each other.

Solution: *Initial state*: 4 queens on the board in random row positions.

Goal state: 4 queens on the board with no pair of queens attacking each other.

By now, you might have realized why I did not define the constraints in the state. Our goal is to find a state that satisfies all the constraints. However, while we are exploring, most of the states we consider will violate at least one constraint. Therefore, there is no point requiring each state to satisfy all the constraints. We will solve 4-queens as an optimization problem by encoding the constraints in the cost function.

Next, let's look at the neighbour relation. I've defined two different neighbor relations. In general, there are many possible neighbour relations for one problem. It is an interesting exercise to think about the difference between these neighbour relations and their impact on the performance of the local search algorithm.

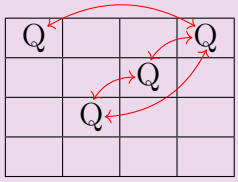
Solution: *Neighbour relation*:

- Version A: move a single queen to a different row in the same column.
- Version B: swap the row positions of two queens.

Finally, here is the cost function. We want to minimize the number of pairs of queens attacking each other, either directly or indirectly. How can two queens attack each other directly or indirectly? Two queens directly attack each other, if they are in the same row or diagonal, and there is no other queen between them. Two queens attack each other indirectly if they are in the same row or diagonal and there is at least one queen between them.

Solution: *Cost function*: the number of pairs of queens attacking each other, directly or indirectly.

For example, the cost of the following state is 4.



This is a complete local search formulation.

Solution:

Variables: x_0, x_1, x_2, x_3 where x_i is the row position of the queen in column i, where $i \in \{0, 1, 2, 3\}$. Assume that there is one queen per column.

Domains: $x_i \in \{0, 1, 2, 3\}$ $\forall i$.

Initial state: 4 queens on the board in random row positions.

Goal state: 4 queens on the board with no pair of queens attacking each other.

Neighbour relation:

- Version A: move a single queen to a different row in the same column.
- Version B: swap the row positions of two queens.

Cost function: the number of pairs of queens attacking each other, directly or indirectly.

Now, let's contrast the two neighbour relations. How are they different? It turns out that,

version A results in a connected search graph, whereas version B produces a search graph with many disconnected components. Once I introduce some local search algorithms, I invite you to think about which neighbour relation works well with which algorithm.

3 Greedy Descent

In this section, I will discuss Greedy Descent, our first local search algorithm.

This algorithm has many other names: hill-climbing, greedy ascent, and iterative best improvement. I will refer to this algorithm as Greedy Descent since our goal is to minimize a cost function.

Greedy Descent works as follows: Start with a random state. At each step, decide if we want to move to a neighbour. If at least one neighbour is an improvement (the neighbour has lower cost than the current state) then move to the best neighbour (the one with the lowest cost). If no neighbour is better than the current state, then the algorithm stops. At this point, the current state is one of the best states in the local neighbourhood.

The phrase below describes the intuition behind "Greedy Descent" nicely. It is modified from a phrase from the Russell and Norvig textbook. Greedy descent is like "descending into a canyon in a thick fog with amnesia". 'Descending' means that we are trying to minimize cost by moving to a neighbour with a lower cost at each step. The 'thick fog' suggests that we can only see the immediate neighbours, and we're not considering any other states beyond the immediate neighbours. Finally, 'amnesia' means loss of memory. 'Greedy descent' only remembers the current state, and it has no memory of where it has been.

Descend into a canyon in a thick fog with amnesia.

3.1 Properties of Greedy Descent

Let's look at some properties of "Greedy Descent". Although "Greedy Descent" looks simple, it performs quite well in practice. Greedy descent often makes progress towards a solution fairly quickly.

For the second property, let's consider the following question. If Greedy Descent has unlimited time to explore the search space, is it guaranteed to find the global optimum? The global optimum means a state with the lowest cost among all the states in the search space. Think about this question for a minute. Then, keep reading for the answer.

Unfortunately, the answer to this question is no. When Greedy Descent terminates, we only know that the current state is the best state among its immediate neighbours. This state is a local optimum: we cannot improve it locally. However, this state might not be the global optimum. If a problem is challenging, then its search space often has a large number of local optimums.

3.2 Definitions of Local and Global Optimums

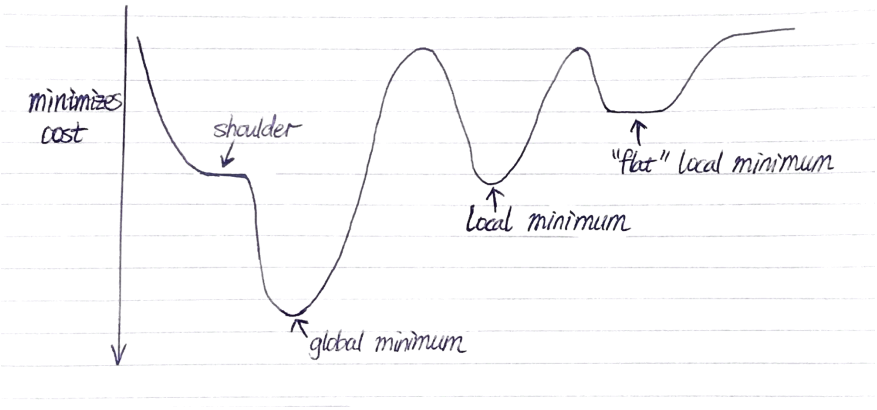
Previously, I introduced Greedy Descent. One major problem with the Greedy Descent algorithm is that it may get stuck at a local optimum and fail to find a global optimum solution.

To understand this property, let's define global and local optimums more formally. The

global optimum is the state that has the lowest cost among all of the states in the search space. A local optimum is a state where no neighbour has a strictly lower cost than the current state.

In other words, a global optimum is the best state in the entire search space, whereas a local optimum is the best state in a local neighbourhood.

Let's visualize global and local optimums in a picture of an one-dimensional search space. The global optimum is the lowest point on the curve.



There are three kinds of local optimums in this picture.

The middle one is a strict local optimum. Being a strict local optimal means that the cost of the state is less than or equal to the cost of all of its neighbours, and then the cost of this state is strictly less than at least one of the neighbours. Therefore, the state is strictly the best state in its neighbourhood.

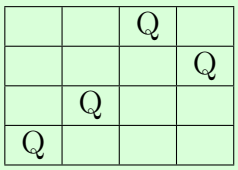
The other two local optimums are in a flat neighbourhood. The left one has a special name called a shoulder. You can tell that the name comes from its shape, which looks like a person's shoulder. If we are on a shoulder and we explore the neighbourhood for long enough, we can escape the shoulder and reach a state with a lower cost. For the other flat local optimum, it is challenging to escape it since the states on both sides have higher costs.

3.3 Practice Questions for Local and Global Optimums

Now that we have defined local and global optimums formally. Let's look at two practice questions. In each question, we are given a state of the 4-queens problem and a neighbour relation, and we are asked to determine whether the state is a local and/or global optimum.

Problem: Consider the following state of the 4-queens problem. Consider neighbour relation B: swap the row positions of two queens. Which of the following is correct?

(A) This state is a local optimum and is a global optimum.
(B) This state is a local optimum and is **not** a global optimum.
(C) This state is **not** a local optimum and is **not** a global optimum.



Solution:

This state is not a global optimum since there are two pairs of queens attack each other.

Is this state a local optimum or not? To answer this, we need to calculate the cost of every neighbour and see if any neighbour has a strictly lower cost than the current state. The current state is 3201 with a cost of 2.

Among the 4 queens, there are 6 ways to choose two queens to swap. For example, if we swap the role positions of the leftmost two queens, we get 2301. If we swap the row positions of the first and the third queens from the left, we get 0231. Let's write down the six neighbours.

Next, let's calculate their costs. Take the state 2301 as an example. There are four pairs of queens attacking each other: 0 and 1, 2 and 3, 0 and 2, and 1 and 3. It has a cost of 4. Let's calculate the costs of all 6 neighbours.

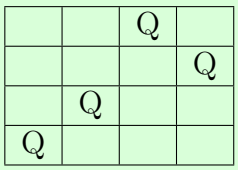
- 2301 (4)
- 0231 (1)
- 1203 (1)
- 3021 (1)
- 3102 (1)
- 3210 (6)

We can see that the current state is not a local optimum since at least one state, 0231, has a strictly lower cost.

The correct answer is (C).

Problem: Consider the following state of the 4-queens problem. Consider neighbour relation A: move a single queen to another square in the same column. Which of the following is correct?

(A) This state is a local optimum and is a global optimum.
(B) This state is a local optimum and is **not** a global optimum.
(C) This state is **not** a local optimum and is **not** a global optimum.



Solution: The state is not a global optimum for the same reason as we discussed in question 1.

Is this state a local optimum or not? Same as before, let's calculate the cost of all the neighbours.

Since the neighbour relation requires us to move one queen, each empty square corresponds one neighbour of the current state. For example, the top left square represents the neighbour state where the leftmost queen is moved to the top left square. For each empty square, I will construct the neighbour in my mind, calculate its cost, and write down the cost of the neighbour in that square.

For example, if we move the leftmost queen to the top left square, then we will have two pairs of attacking queens. The cost of this neighbour is 2. If we move the third queen down by one, we will have 4 pairs of attacking queens, 0 and 1, 1 and 2, 0 and 2, and 2 and 3. So the cost of this neighbour is 4.

Let's calculate the costs of all the neighbours. The current state has a cost of 2 and every neighbour's cost is 2 or more. Therefore, the current state is a local optimum.

2	2	Q	4
3	3	4	Q
3	Q	3	2
Q	3	3	2

The correct answer is (B).

3.4 Escaping Flat Local Optimums

As you have seen, a major problem with Greedy Descent is that it may get stuck at a local optimum. What can we do to overcome this problem? Let's explore a few strategies for escaping local optimums.

Let's think about how we can escape a flat local optimum, in particular, a shoulder. When Greedy Descent reaches a flat area, that is, when all the neighbours have the same cost as the current state, it terminates right away. The reason is that by our definition, a state within a flat neighbourhood is a local optimum.

In this case, one strategy is to keep moving. This strategy is called sideway moves. The algorithm is allowed to move to a neighbour with the same cost as the current state. One problem with allowing sideway moves is that the algorithm might not terminate. Imagine that we are exploring a flat area. Since the algorithm does not remember where it has been, it could be moving in the flat area forever. So, if we allow sideway moves, we need to put a limit on it. Typically, we will specify that the algorithm terminates after making a maximum number of consecutive sideway moves.

Even if we allow sideway moves, Greedy Descent may be exploring the flat area inefficiently. Recall that Greedy Descent does not have any memory. It may be going in circles and not realizing it. To make the algorithm more efficient, we can give it some short-term memory, in the form of a tabu list. A tabu list records a list of recently visited states so that Greedy Descent does not return to these states immediately. However, the tabu list must have a limited size. If the list is filled up, we will have to replace some old states in the list with new ones, and it is still possible for Greedy Descent to explore states that it has visited before.

3.5 Performance of Greedy Descent with Sideway Moves

You might be wondering, do these strategies really help Greedy Descent perform better? Let's look at some statistics on how Greedy Descent performs when we allow sideway moves.

Consider the 8-queens problem. It has 17 million states.

Without sideway moves, Greedy Descent can solve 14% of the problem instances. With at most 100 consecutive sideway moves, Greedy Descent can solve more than 90% of the problems.

However, there is a trade-off. Without sideway moves, Greedy Descent terminates fairly quickly, in 3-4 steps on average. With sideway moves, Greedy Descent spends a lot more time exploring flat areas. On average, the algorithm takes 21 steps until success and 64 steps until failure.

The trade-off is: Sideway moves allows us to solve more problem instances at the cost of taking much longer time before the algorithm terminates.

4 Greedy Descent with Random Moves

In the previous section, I discussed two strategies for escaping flat local optimums — sideway moves and tabu list. In this section, let's look at how we can escape strict local optimums using random moves.

4.1 Two Types of Random Moves

Let's start by looking at two types of random moves: random restarts, and random walks.

A random restart is a potentially big jump in a search space. If the algorithm is stuck in a particular region, we will generate and move to a random state in the space. Adding random restarts will improve the property of Greedy Descent significantly. I will discuss this in more detail in this lecture.

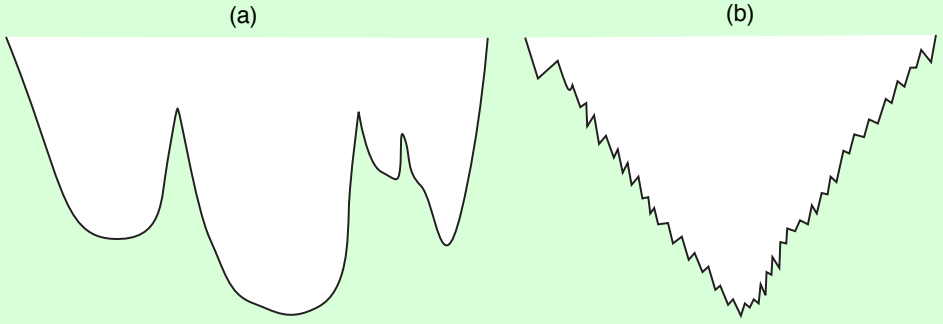
The second type of random move is a random walk. A random walk says that we will move to a random neighbour in our local neighbourhood. By doing this, it is possible for us to move a neighbour that is worse than the current state. Greedy descent does not allow random walks. In the next section, I will discuss another algorithm called simulated annealing, which makes use of random walks to explore the search space.

Let's think about the two types of random moves. When is a good idea to use one random move versus the other?

Problem:

Which random move is better for search space (a) or (b)?

(A) Random restarts
(B) Random walks



Solution:

The correct answers are: For search space A, random restarts is a better choice. For search space B, random walks is a better choice. Why are these the case?

The main difference between the two search spaces is that A is smooth, where B is jagged. A has few local optimums, whereas B has a lot of local optimums.

In the smooth search space A, Greedy Descent will perform quite well. Greedy descent will take us to the local optimum quickly since there is a clear direction towards the local optimum. Random walks are not helpful since they may take us farther away from the local optimum. However, once we reach a local optimum and want to explore other parts of the search space, only a random restart will allow us to perform a big jump, get us out of the local optimum and possibly take us to a new region, where there may be a better local optimum. Therefore, for space A, random restarts are a better choice.

Let's look at search space B. Since there are lots of local optimums. Greedy descent will get stuck in one very quickly. After that, if we perform a random restart, we will jump to another region. Unfortunately, this new region is still bumpy. Greedy descent will once again get stuck right away. As you can see, a big jump is not useful. No matter which local neighbourhood we are in, we will get stuck right away. However, a random walk can be helpful. Moving randomly in a local neighbourhood is a better way of getting out of a local optimum and possibly make progress towards the global optimum. Therefore, for space B, random walks are a better choice.

Intuitively, a random restart is a global random move, whereas a random walk is a local random move.

4.2 Greedy Descent with Random Restarts

Let's look at how random restarts can help Greedy Descent escape strict local optimums. Recall that, Greedy Descent can find a local optimum fairly quickly. However, the problem is that, the first local optimum reached by Greedy Descent is often not the global optimum.

What if we combine Greedy Descent with random restarts? Each time, we will execute Greedy Descent until it finds a local optimum and terminates. After that, we will generate a random state and start Greedy Descent again from the random state. After running Greedy Descent for a number of times, we return the best state among all the local optimums found.

This quote is a great intuitive description of Greedy Descent with random restarts:

> If at first you don't succeed, try, try again.

This is one of my favourite quotes. Also, this quote is a great way to approach life. If you have an important goal and you don't succeed the first time, then keep trying!

How does random restarts improve the properties of Greedy Descent? Recall that, if we perform Greedy Descent without random restarts, even if we have unlimited time, Greedy Descent is complete — it is not guaranteed to find the global optimum. In fact, Greedy Descent will not find the global optimum most of the time. Does the answer change if we

perform Greedy Descent with random restarts? Think about this question for a minute. Then, keep reading for the answer.

The correct answer is yes. Greedy descent with random restarts is complete with probability 1. Given enough restarts, Greedy Descent is guaranteed to find the global optimum. There is a simple reason for this: Given enough time, the algorithm will eventually generate the global optimum as the initial state, and it will terminate immediately.

Problem:

In the past, students has raised an interesting question. We know that Greedy Descent can determine whether a state is a local optimum by comparing the state with its neighbour. The question is: how does Greedy Descent determine whether a state is a global optimum or not? I'll leave this as a thought question for you. The answer is different depending on whether we consider a constraint satisfaction problem or a pure optimization problem.

5 Simulated Annealing

5.1 Brief Recap

Previously, I talked about how adding random restarts can improve the properties of Greedy Descent. With random restarts, Greedy Descent will find the global optimum with probability approaching 1.

Why is it a good idea to combine Greedy Descent with Random Restarts? Well, Greedy Descent focuses on optimization — it tries to improve the state as much as possible and as quickly as possible. Another name for this is exploitation: if we are already good at doing something, we should keep doing it. On the other hand, random restart allows us to explore our search space. We can jump to a completely different part of the search space and hope that we will find a state with a lower cost. Greedy Descent with random restarts has good properties since it combines the ideas of exploration and exploitation.

5.2 Introducing Simulated Annealing

Simulated annealing combines exploration and exploitation in another way. 'Annealing' is a term from physics. You must have read about or seen the process of forging a sword, or blowing glass. We start by heating up the metal Then, we will cool the metal down slowly, to put it into a better shape and to make it strong.

In the simulated annealing algorithm, we will keep track of a temperature, initialize it to a large value and decrease this temperature slowly.

At each step, we will choose a random neighbour of the current state. Note that this is a random neighbour, not the best neighbour. There are two cases.

Case 1: The neighbour is an improvement. That is, the neighbour has lower cost than the current state. We will move to the neighbour since we want to minimize the cost of the current state.

Case 2: What if the neighbour is not an improvement? Well, it is not a bad idea to move to such a neighbour some of the time. Remember that Greedy Descent has a bad property — it can get stuck at a local minimum. If we're willing to move to a worse neighbour sometimes, this is one way we can get out of a local minimum. So, if the neighbour is not an improvement, we will move to it with some probability.

This probability depends on two things: the current temperature and how much worse the neighbour is compared to the current state. If the random draw based on the probability decides that we should not move, we will stay put and choose another random neighbour.

5.3 The Probability of MOving to a Worse Neighbour

How can we determine the probability of moving to a worse neighbour? Let A be the current state and A' be the neighbour that's worse than the current state. Let ΔC represent the cost difference between A' and A. ΔC should be positive since A' has a higher cost than A.

T is the current temperature. We will move to A' with a probability of e to the power of negative ΔC divided by T.

This function is called the Gibbs distribution or Boltzmann distribution. It appears in several other fields such as physics. This function is also related to the sigmoid or the logistic function, which you will encounter when learning neural networks in machine learning.

Let's work on two questions, which will help us understand how the probability depends on the current temperature and the cost difference.

Problem: A is the current state and A' is the worse neighbour.

Let $\Delta C = \text{cost}(A') - \text{cost}(A)$.

As T decreases, how does the probability of moving to the worse neighbour ($e^{-\frac{\Delta C}{T}}$) change?

(A) As T decreases, we are more likely to move to the neighbour.

(B) As T decreases, we are less likely to move to the neighbour.

Solution: The correct answer is (B).

Intuitively, moving to a worse neighbour is a risky move. As temperature decreases, we want to be less risky, so we should be less likely to move to the neighbour.

Let's verify this mathematically. We know that $\Delta C > 0$ and $T > 0$, so $\frac{\Delta C}{T} > 0$. Then as T decreases, $\frac{\Delta C}{T}$ increases, $-\frac{\Delta C}{T}$ decreases, and finally $e^{-\frac{\Delta C}{T}}$ decreases.

Another way is to just plug in some values. Let's take $\Delta C = 10$ and see how the probability changes as we decrease T from 100 to 10:

$$T = 100 \qquad e^{-10/100} = e^{-0.1} \approx 0.9$$
$$T = 10 \qquad e^{-10/10} = e^{-1} \approx 0.36$$

Again, the probability decreases.

Problem: A is the current state and A' is the worse neighbour.

Let $\Delta C = \text{cost}(A') - \text{cost}(A)$.

As ΔC increases (the neighbour becomes worse), how does the probability of moving to the worse neighbour ($e^{-\frac{\Delta C}{T}}$) change?

(A) As ΔC increases, we are more likely to move to the neighbour.

(B) As ΔC increases, we are less likely to move to the neighbour.

Solution: The correct answer is (B).

Intuitively, moving to a worse neighbour is already a risky move. If a neighbour gets even worse, then moving there becomes riskier, so we should be less likely to move there.

Lets verify this mathematically again. We know that $\Delta C > 0$ and $T > 0$, so $\frac{\Delta C}{T} > 0$. Then as ΔC increases, $\frac{\Delta C}{T}$ increases, $-\frac{\Delta C}{T}$ decreases, and finally $e^{-\frac{\Delta C}{T}}$ decreases.

Again, we could also use some numbers. Let's take $T = 10$ and see how the probability changes as we increase ΔC.

$$\Delta C = 10 \qquad e^{-10/10} \approx 0.36$$
$$\Delta C = 100 \qquad e^{-100/10} \approx 0.000045$$

Again, the probability decreases.

The takeaway is that moving to a worse neighbour is risky. As the temperature decreases and as the neighbour gets worse, we want to move there with a smaller probability.

5.4 Algorithm

Here is a pseudo-code description of the simulated annealing algorithm.

Algorithm 1 Simulated Annealing

```
current ← initial-state
T ← a large positive value
while T > 0 do
    next ← a random neighbour of current
    ΔC ← cost(next) − cost(current)
    if ΔC < 0 then
        current ← next
    else
        current ← next with probability p = e^(−ΔC/T)
    decrease T
return current
```

First, choose the initial state randomly. Initialize the temperature to a large positive value. We'll decrease the temperature slowly. As long as the temperature is above some minimum value, say zero, we'll do the following: - Choose a random neighbour. - Calculate the cost difference between the neighbour and the current state. - Decide whether we want to move

to the neighbour or not. - If the neighbour is an improvement, we will move to it for sure. - If the neighbour is worse, we will move to the neighbour with some probability.

When the temperature reaches the minimum value, we will stop and return the current state.

One thing you might be wondering is: how can I implement this probability? Here is a common trick. Draw a random number between 0 and 1. If this number is smaller than the probability, it is a yes and we will move to the neighbour. Otherwise, it is a no and we will stay put and choose another random neighbour.

5.5 Annealing Schedule

The algorithm decreases the temperature at each step. How exactly do we do this? The way we decrease T is called the annealing schedule. In practice, we want to decrease the temperature very, very slowly. If the temperature decreases slowly enough, then simulated annealing is guaranteed to find the global optimum with a probability approaching 1. This is a great property and similar to what we have for Greedy Descent with random restarts.

Finding a good annealing schedule is an art rather than a science, and it highly depends on the exact search problem that we're trying to solve. One of the most widely used annealing schedules is geometric cooling. Geometric cooling works by multiplying T by a value that is less than 1 but very close to 1. For example, multiply T by 0.99 at every step. You can use many other smooth functions as the annealing schedule such as any linear, logarithmic, or exponential functions.

5.6 An Analogy

In this section, I want to give you some intuitions behind simulated annealing.

First, let me share an analogy to help you understand simulated annealing. Let's imagine a big surface, one that we can still hold and control. The surface has mountains and valleys. We drop a tennis ball onto the surface. Our goal is to get this tennis ball into the deepest valley on the surface. We cannot touch the ball, but we can shake the surface. How do we do that? Gravity's our friend. Gravity pulls the ball downward at any time. This represents the force that is optimizing for us. However, the ball might get into a shallow valley first and get stuck there. That's a local minimum. To avoid this, we will start by shaking the surface really intensely — resembling the case when we have a high temperature. The ball will bounce around a lot on the surface. During the exploration phase, the ball may get to many different parts of the big surface. As time goes on, we will decrease the shaking slowly. Hopefully, when the shaking stops, the ball will rest in one of the relatively deep valleys on the surface.

Now, let me be a bit philosophical. Simulated annealing is like life. It's a perfect example of an exploration versus exploitation trade-off. When the temperature is high, what does that represent? You are young and energetic and life is full of possibilities. In this phase, you try a lot of different things. What does trying a lot of different things mean? It means making

a sub-optimal move. At this stage, you're not optimizing that much. Instead, you are trying out as many things as you can.

As you get older, you gradually start to optimize more, for your goals and dreams. Hopefully, you find something you are good at, something you enjoy and you are passionate about. You gradually get into the exploitation phase where you are trying to improve and optimize the current state of your life.

Simulated annealing is like life in the sense that in the earlier phase we tend to explore more, where in the later phase we tend to optimize more. This is the trade-off between exploration and exploitation.

Here's something for you to think about: Which phase of life are you in right now? Are you still exploring a lot, or are you mostly just optimizing?

Here are my two cents: I believe that we should always keep a little bit of adventure in our heart because life is full of possibilities. I hope that no matter how old you are, which stage of life you are in, that you always have the courage to explore and to try and learn new things.

6 Population-Based Algorithms

So far, the local search algorithms only remember a single state. The advantage is that these algorithms require very little memory. The disadvantage is that they can only keep track of one state at a time. Perhaps, if they could remember multiple states, they would have a better chance at finding the best state among them.

Let's explore an idea called population-based local search algorithms where the algorithm remembers multiple states at each time step.

6.1 Beam Search

Our first population-based algorithm is called Beam Search. Beam search remembers a population of k states, where $k > 1$.

How do we update the population of k states? At each step, we will look at all the neighbours of these k states, and we will choose the best $k$states among all the neighbours to be the population at the next time step.

The value of k affects Beam Search in several ways. It affects the amount of memory we need. The larger the k, the more memory we need to store the population.

The value of k also controls the level of parallelism. A larger k makes the search more parallel since we are simultaneously looking at the neighbours of a larger number of states in different regions of the search space.

Let's think about three questions. They can help you think more about the properties of Beam Search.

Problem: What does Beam Search look like when $k = 1$?

Solution:

When $k = 1$, Beam Search is essentially Greedy Descent. It remembers one state and moves to the best neighbour at each time step.

Problem: How is Beam Search different from running Greedy Descent with k random restarts in parallel?

Solution: How is Beam Search different from Greedy Descent with k random restarts in parallel? If we run k random restarts, these random restarts are independent from each other. Each search updates its state independent of the other states. However, with Beam Search, we are choosing the best k states among all the neighbours of the k current states. Intuitively, the k states are not operating independently and these is some sort of communication among them. For example, if one of the k states, called X,

has some really good neighbours, then it is possible that the next population consists of only state X's neighbours. It is as if state X is waiving its hands to the other states and saying that "Look! I'm in a really good region! Come over here and join me." Therefore, k random restarts in parallel are independent searches, whereas the k states in Beam Search are not independent.

Problem: Can you think of any problems with Beam Search?

Solution: Can you think of any problems with Beam Search? The example I gave for question 2 may give you some ideas. If one of the k states is in the region we have found so far, Beam Search will quickly concentrate its effort in this region only. Doing this is great for optimization, but not so great for exploration. Whenever we find a good region in the search space, we tend to cluster there and ignore other parts. This causes our population to lose diversity very quickly. This problem leads to our next algorithm called Stochastic Beam Search, which tries to improve upon Beam Search to maintain diversity in the population.

6.2 Stochastic Beam Search

Stochastic Beam Search is quite similar to Beam Search. It remembers a population of k states. The main difference is how it updates the k states. While both algorithms will look at all the neighbours of the k states, Beam Search chooses the next k states deterministically. In contrast, Stochastic Beam Search chooses the next k states probabilistically. The probability of choosing each state is inversely proportional to the cost of the state. The lowest the cost of the state, the higher the probability of choosing the state. A crucial difference is that Beam Search will choose the best state only, whereas Stochastic Beam Search may choose every state with a positive probability.

For Stochastic Beam Search and the genetic algorithm, we want to convert the goodness of a state to a probability. So it is more convenient to talk about fitness rather than cost. Fitness is the negative of cost. Minimizing cost is equivalent to maximizing fitness. If we convert each state's cost to each state's fitness, then the probability of choosing a neighbour is proportional to the state's fitness.

Stochastic Beam Search is better than Beam Search at maintaining diversity in the population. Since there is a positive probability of choosing every neighbour, the search is less likely to cluster in the best region found so far. This gives us a chance to explore different parts of the search space.

In a sense, Stochastic Beam Search mimics the process of natural selection. Imagine that each state is an organism. The state's neighbours are potential offspring. The survival of the potential offsprings depends on their fitness. Some will survive, and others will not. The offspring that survive will make up the next population or the next generation of organ-

isms. Intuitively, Stochastic Beam Search resembles asexual reproduction because each state produces offspring without mixing with other states.

6.3 The Genetic Algorithm

Finally, let's look at the star of this lecture: the genetic algorithm. The genetic algorithm draws its ideas from biology and evolution. It is similar to Stochastic Beam Search: It maintains a population of k states and chooses the next k states probabilistically. However, this probabilistic process is different for Stochastic Beam Search and the genetic algorithm are different. Stochastic Beam Search resembles asexual reproduction whereas the genetic algorithm represents sexual reproduction.

To produce each state in the next population, we start by choosing two parent states from the current population. The probability for choosing a state to be a parent is proportional to the fitness of the state. The higher the fitness of a state, the higher probability of choosing the state for reproduction. Note that, a state can be chose multiple times for reproduction. Once two parent states are chosen, they are mixed in a cross-over process to produce a child state. There are many ways of performing cross-over. Typically, the child will keep a portion from each parent. After cross-over, the child state may mutate with a probability, mimicking a random mutation in evolution. For example, we can choose a portion of the child state and randomly change it to some other value. This final child state is one member of the new population. We will repeat these steps until we have produced k states. These k states make up the population for the next time step.

We will repeat this process until some stopping criteria are satisfied. One stopping criteria could be at least one state in the population has a fitness that is greater than some desired threshold value.

Please check out a separate PDF file for an example of executing the genetic algorithm on the 8-queens problem.

6.4 A Fun Genetic Algorithm Car Simulator

Check out this genetic algorithm car simulator. It uses a simple genetic algorithm to evolve random two-wheeled shapes into cars over generations.

https://rednuht.org/genetic_cars_2/

6.5 Greedy Descent v.s. The Genetic Algorithm

So far, I have discussed a few different local search algorithms. At the surface, some of them appear quite different. For example, you might think that Greedy Descent appears to be quite different from the genetic algorithm. But are they really that different? Let's do a quick comparison between them. The purpose of this discussion is to help you realize that these two algorithms have some highly similar components and are not that different.

Recall that many local search algorithms try to trade off exploration and exploitation (or

exploration and optimization). Let's consider Greedy Descent and genetic algorithm and think about how each performs the trade-off.

Problem: How does each algorithm explore the search space?

Problem: How does each algorithm optimize the quality of the population?

Pause for a few minutes and come up with your answers. Then, keep reading.

For exploration, Greedy Descent relies on the neighbour relation. It considers moving to a neighbour of the current state. The generic algorithm generates a neighbour through cross-over and mutation. Starting with two states in the current population, the genetic algorithm produces a new state by cross-over and modifies the state with a small probability using a random mutation. The cross-over and mutation process resembles a randomized neighbour relation. Hopefully, the child state is quite different from the two parents and allows the genetic algorithm to explore a new region in the search space.

For optimization, Greedy Descent optimizes the quality of the current state by moving to the best neighbour. What about the genetic algorithm? The genetic algorithm optimizes the population by choosing the states for reproduction based on their fitness. The higher the fitness of a state, the more likely the state is chosen for reproduction. The idea is that, if the two parent states have relatively high fitness, then the child state will likely have a higher fitness as well.

This scheme reminds me of a Chinese animation that I watched recently. It's called Ling Long (spirit cage: incarnation). In a dystopian world, humans were nearly wiped out. The remaining humans had to form a society with several new rules. Relationships are banned. More importantly, only people with superior genes are allowed to have babies. This idea is very similar to the genetic algorithm. This idea is interesting but controversial as there are obvious ethical issues with using this scheme for the human society.

7 Practice Problems

1. When executing a local search algorithm, we can use random walks or random restarts to escape from local optima.

   Give an example of a search space such that it is better to perform random walks than random restarts when executing a local search algorithm on the search space.

   Give an example of a search space such that it is better to perform random restarts than random walks when executing a local search algorithm on the search space.