

Local Search

CMPUT 366: Intelligent Systems

P&M §4.7

Logistics & Assignment #1

- **Midterm** is **March 11**
- **Assignment #1 was released Wednesday**
See eClass
 - Due **February 4** at 11:59pm
- Office hours have begun!
 - Not mandatory; for getting help from TAs
 - There are no labs for this course

Recap

- **Graph search problems** are an extremely general encoding for choosing a **sequence of actions** from a start state to a goal state
- Using **heuristic functions** can speed this process up
 - **A* search** is optimal but space-intensive
 - **Branch & bound depth-first search** is optimal and space efficient, but needs a good starting bound
- Varying the direction of search can exploit mismatches in forward and reverse **branching factors**

Lecture Outline

1. Recap & Logistics
2. Local Search
3. Hill Climbing
4. Randomized Algorithms

Searching for Goal Nodes

Sometimes, we know how to **recognize** a goal node, but not how to **construct** one.

Example (SAT problem): Given a Boolean formula,

$$P(X) = (X_1 \wedge X_2 \wedge \neg X_3) \vee \dots \vee (\neg X_{k-2} \wedge \neg X_{k-1} \wedge X_k),$$

is there an assignment of truth values to the variables X_i that makes the formula true?

- **State** is the values of the different variables
- **Easy to recognize** when we've succeeded, but computing a "satisfying assignment" is **NP-complete** in general
- **SAT** is an example of a **constraint satisfaction problem**

Searching for Goal Nodes

We can encode SAT as a graph search problem (assignments as states, variable value changes as actions), *but*:

1. The **space is too big** to explore exhaustively
 - **Question:** How many states are there in a SAT problem with k variables?
 - Industrial SAT problems routinely have **hundreds of thousands** of variables
2. We don't care about the **sequence of actions**
 - Once we have a satisfying assignment, we are done

Local Search

- **Idea:** start from a random assignment, and then search around in the space of possible assignments
- Need not keep track of the sequence of moves that we took
- Intuitively:
 1. Select an assignment of a value to each variable
 2. Repeat:
 - (i) Select a variable to change
 - (ii) Select a new value for that variable
 3. until a satisfying assignment is found

Local Search Problem

Definition: Local Search Problem

- A **constraint satisfaction problem**: A set of **variables**, **domains** for the variables, and **constraints** on their joint assignment.
- **Neighbours function**: $neighbours(n)$
 - Maps from a node n to a set of "similar" nodes
- **Score function**: $score(n)$
 - Evaluates the "quality" of an assignment

Questions:

1. What are the **nodes**?
2. What are the **goal nodes**?

Neighbourhoods

- In previous graph search problems, the **successor function** represents states that can be **reached** from a given state by taking some actual action
 - In local search problems, the **neighbours function** is a **design decision**
 - We choose actions that will help us efficiently **explore the space** rather than trying to represent **actual actions**
- Usually the **neighbourhood** is states that differ in **small ways** from the current state (variable assignment)
 - E.g.: Assignments that differ in k different variables, possibly by a small amount
- **Question:** What might be a good **neighbourhood** function for **SAT**?

Heuristics vs. Scores

- Previously, the heuristic was **optional**, for improving efficiency
- In local search problems, the **score** function is **required**
 - The state space is **too big** to exhaustively explore, so uninformed search is not an option
 - Sometimes we don't even have a goal, we just want to **maximize the quality** of the state
- Example scores: number of unsatisfied clauses (in SAT); number of violated constraints (in CSP)

Generic Local Search Algorithm

Input: a constraint satisfaction problem; a *neighbours* function;
a *score* function to maximize; a *stop_walk* criterion

current := random assignment of values to variables

incumbent := *current*

repeat

if *incumbent* is a satisfying assignment:

return *incumbent*

if *stop_walk*():

current := new random assignment of values to variables

else:

select a *current* from *neighbours*(*current*)

if *score*(*current*) > *score*(*incumbent*):

incumbent := *current*

until termination

Hill Climbing

- **Idea:** Select the neighbour with the **highest score**
 - This is called an **improving step**
 - If no improving steps available, **halt** and return *incumbent*
- We'll move toward the best solution once we are close enough
- This algorithm is called **hill climbing**:
 - It seeks the **highest point** on the scoring function's graph
 - It moves only **uphill** (i.e., it makes only improving steps)

Hill Climbing Algorithm

Input: a constraint satisfaction problem; a *neighbours* function; a *score* function

current := random assignment of values to variables

incumbent := *current*

repeat

if *incumbent* is a satisfying assignment:

return *incumbent*

if *False*:

current := new random assignment of values to variables

else:

current := *n* from *neighbours(current)* with maximum *score(n)*

if *score(current)* > *score(incumbent)*:

incumbent := *current*

else:

return *incumbent*

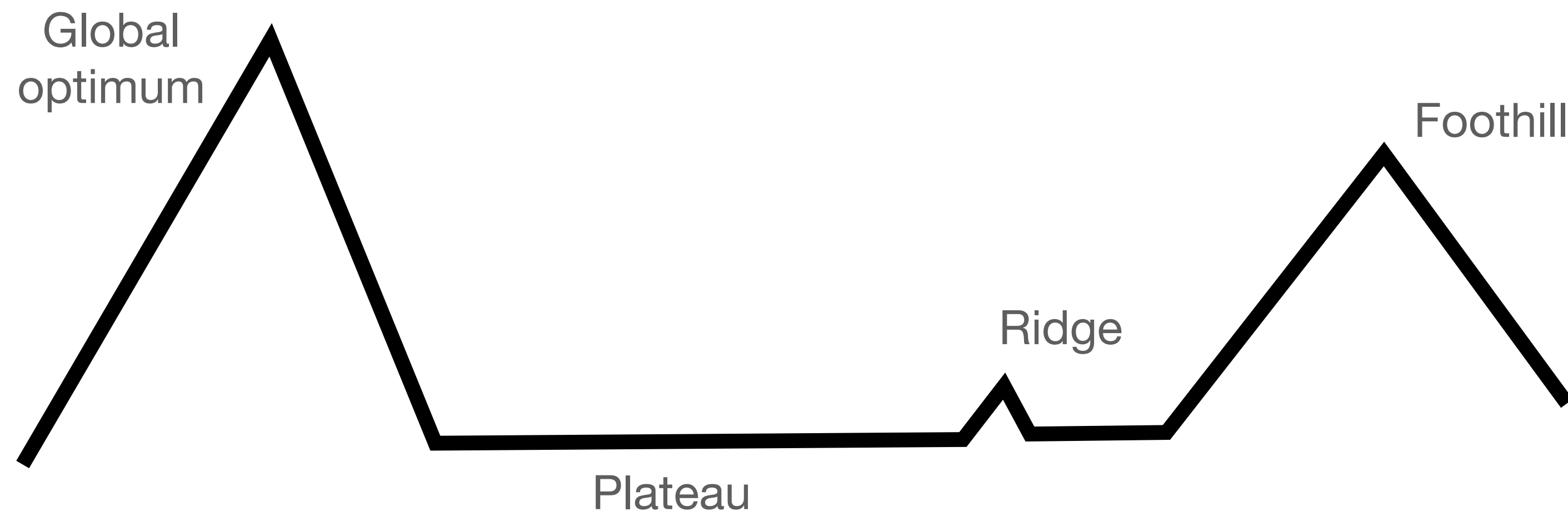
until termination

Questions:

1. Is hill climbing **complete**?
2. Is hill climbing **optimal**?

Hill Climbing Problems

1. **Foothills:** **Local maxima** that are not global maxima
2. **Plateaus:** Regions of the state space where the **score is uninformative**
3. **Ridges:** Foothills that would not be foothills with a **larger neighbourhood**
4. **Ignorance of the global optimum:** Unless we reach a satisfying assignment, we cannot be sure that an optimum returned by local search is the **global optimum**.



Randomized Algorithms

- Adding **random moves** can fix some hill climbing problems
- Two main kinds of random move:
 1. **Random restart:** Start searching from a **completely** random new location
 2. **Random step:** Choose a random **neighbour**
- **Stochastic random search:** Add both kinds of random moves to hill climbing

Stochastic Local Search

Input: a constraint satisfaction problem; a *neighbours* function; a *score* function to maximize; a *stop_walk* criterion; a *random_step* criterion

current := random assignment of values to variables

incumbent := *current*

repeat

if *incumbent* is a satisfying assignment:

return *incumbent*

if *stop_walk*():

current := new random assignment of values to variables

else if *random_step*():

current := a random element from *neighbours*(*current*)

else:

current := *n* from *neighbours*(*current*) with maximum *score*(*n*)

if *score*(*current*) > *score*(*incumbent*):

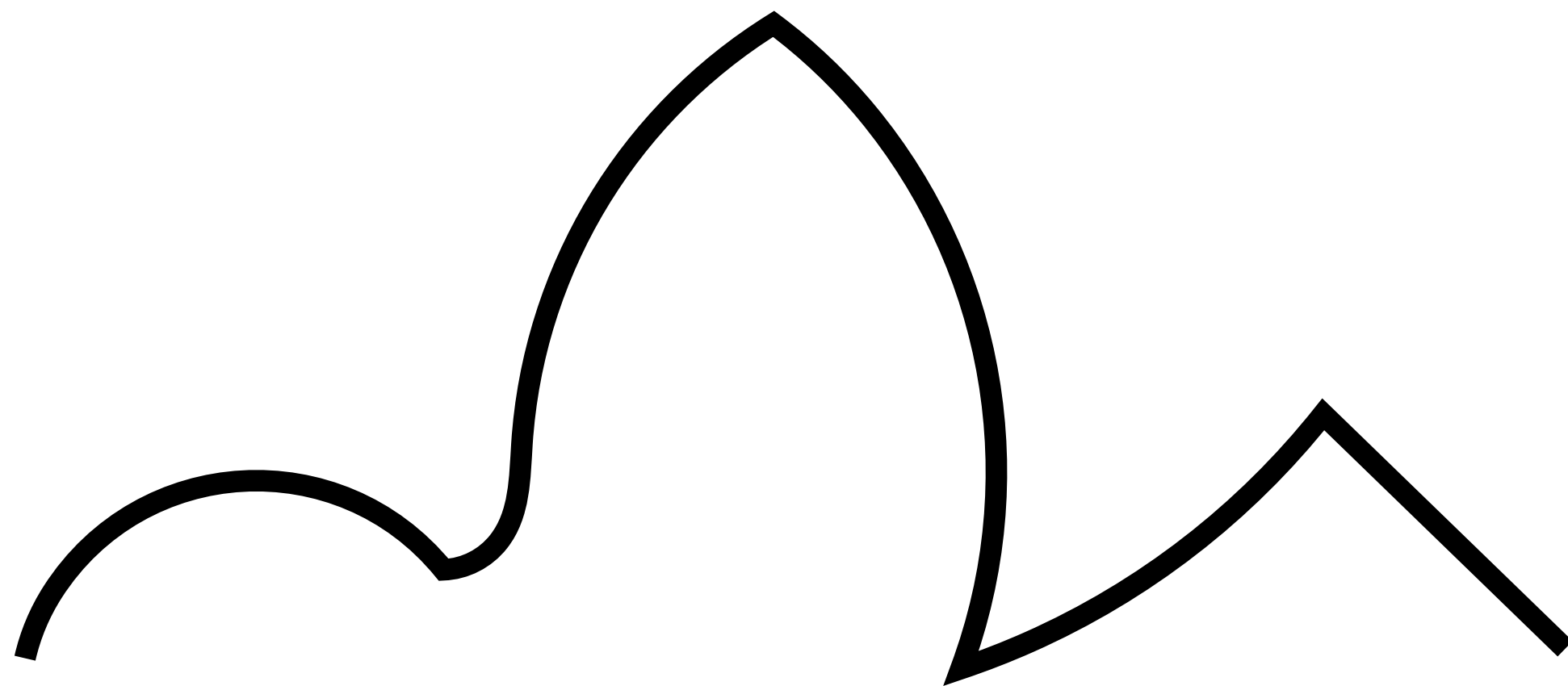
incumbent := *current*

Questions:

1. Is stochastic local search **complete**?
(**Why?**)
2. Is stochastic local search **optimal**?
(**Why?**)

Two Examples

- Consider two partial algorithms:
 1. Hill climbing plus **random restart**
 2. Hill climbing plus **random steps**
- **Question:** Which finds the maximum most easily on each of these two search spaces? Why?



Simulated Annealing

- **Idea:** **Start out** by searching pretty randomly, but become **more directed**
 - **Intuition:** Move to a good neighbourhood quickly, then search intensively in that neighbourhood
- Maintain a "**temperature**" T
- Choose new nodes more randomly at higher temperatures;
Gradually decrease the temperature (according to a **cooling schedule**)
- At each step:
 1. Randomly choose a neighbour *new*
 2. **Always accept** (i.e., assign to *current*) if $score(new) > score(current)$
 3. Else, accept with **probability**

$$e^{[(score(new) - score(current))/T]}$$

Summary

- For some problems, we only care about finding a **goal node**, not the actions we took to find it
- **Local search:** Look for goal states by iteratively moving from a **current state** to a **neighbouring state**
 - **Hill climbing:** Always move to the **highest-score** neighbour
 - **Random step:** Sometimes choose a **random** neighbour
 - **Random restart:** Sometimes start again from an **entirely random** state
 - **Simulated annealing:** Random moves start very **random**, become more **greedy** over time