Part 1 Conclusion: Solving Problems with Search

[Acknowledgment: Some Slides adapted from Dan Klein and Pieter Abbeel]

http://ai.berkeley.edu.]

A*… in 3 lines ☺

```
def Astar(problem):
    """Search the shallowest nodes in the search tree first."""
    return astar_search(problem, util.PriorityQueue, heuristic)
```

```
def heuristic(State s, Problem p):
      return ….
```

```
…
fcost = gcost + heuristic(node.state, problem)
    fringe.push(node, fcost)
```

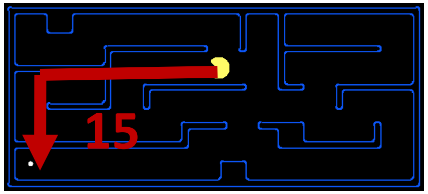
Admissible Heuristics

- A heuristic h is *admissible* (optimistic) if:

$$0 \leq h(n) \leq h^*(n)$$

where $h^*(n)$ is the true cost to a nearest goal

- Examples:



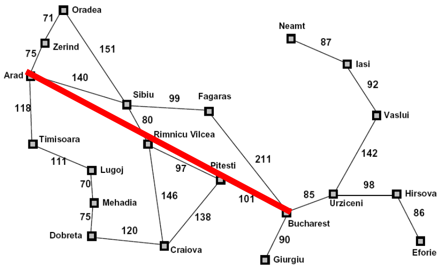
- Coming up with admissible heuristics is most of what's involved in using A* in practice.

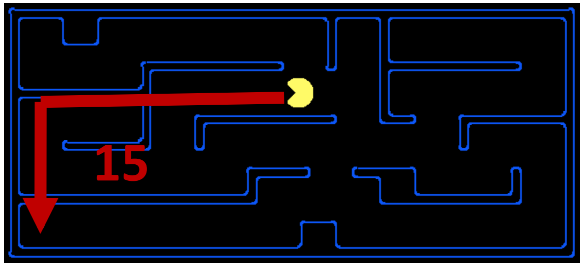
Creating Admissible Heuristics

- **Most of the work in solving hard search problems optimally is in coming up with admissible heuristics**

- Often, admissible heuristics are solutions to *relaxed problems,* where new actions are available

366





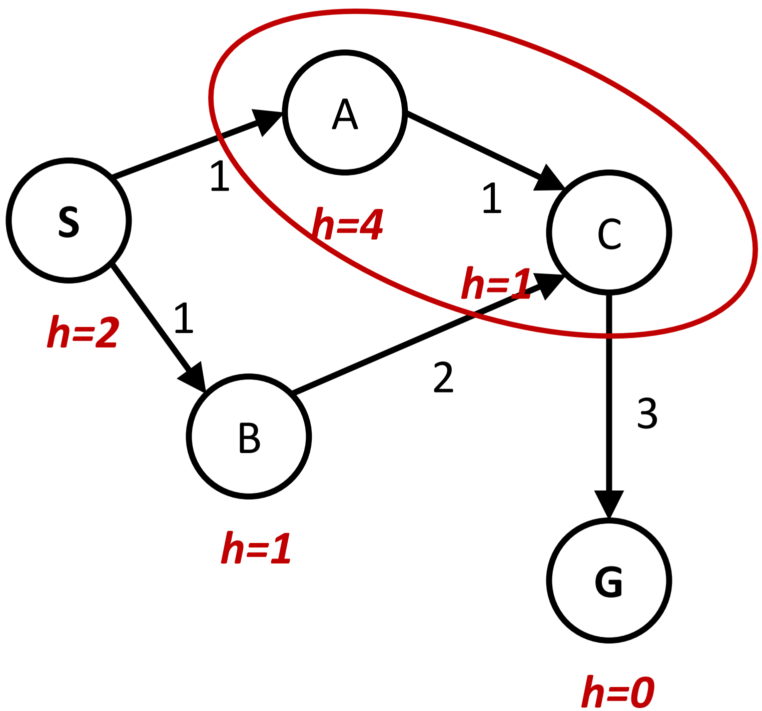
- Inadmissible heuristics can be useful too!

Designing Heuristics

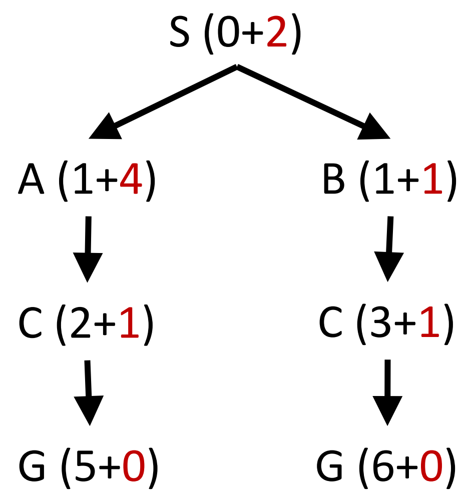
- A good heuristic is:
  - ✓ Admissible (optimistic)
  - ➢ Consistent (non-decreasing)
  - ✓ "Accurate"

A* Graph Search Gone Wrong?

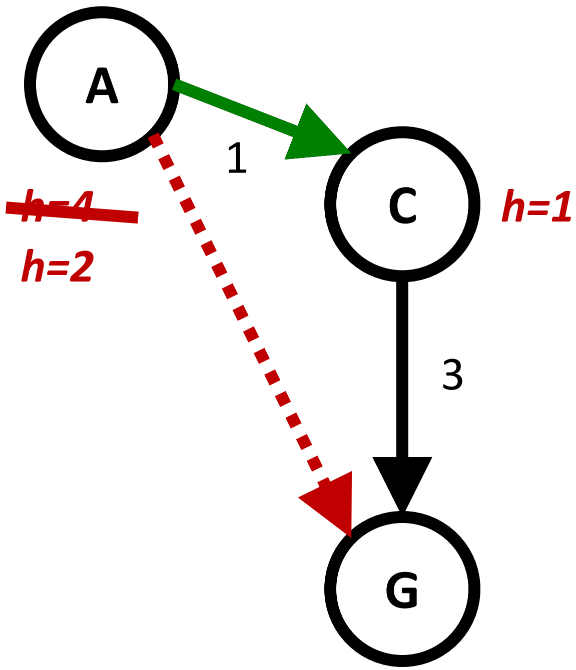
State space graph



Search tree



Consistency of Heuristics



- Main idea: estimated heuristic costs ≤ actual costs
  - Admissibility: heuristic cost ≤ actual cost to goal

    h(A) ≤ actual cost from A to G

  - Consistency: heuristic "arc" cost ≤ actual cost for each arc

    h(A) – h(C) ≤ cost(A to C)

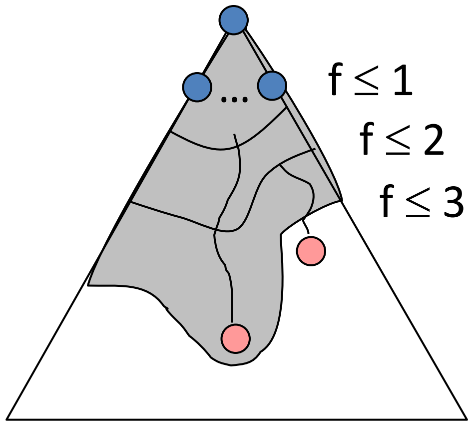
- Consequences of consistency:
  - The f value along a path never decreases

    h(A) ≤ cost(A to C) + h(C)

  - A* graph search is optimal

Optimality of A* Graph Search

- Sketch of proof: consider what A* does with a **consistent** heuristic:
  - **Fact 1:** In tree search, A* expands nodes in increasing total f value (f-contours)
  - **Fact 2:** For every state s, nodes that reach s optimally are expanded before nodes that reach s suboptimally
  - Result: A* graph search is optimal



Example: Heuristics for Motion Planning

- Robot motion: many moving (body) parts
- What's the most efficient way to accomplish goal?

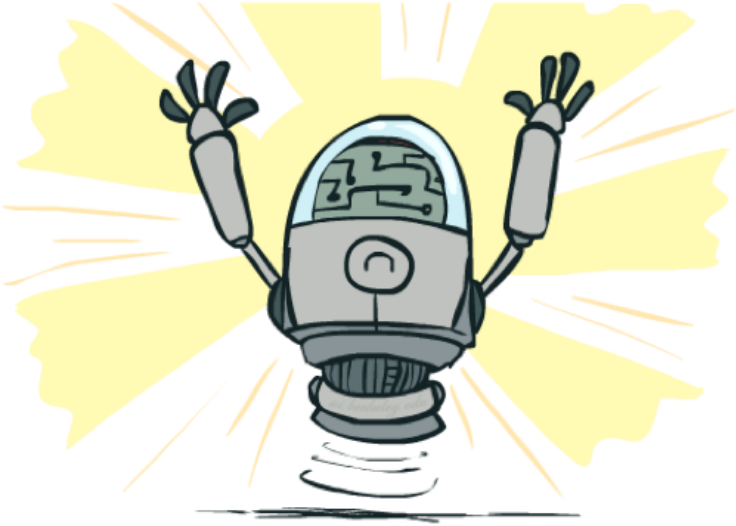
https://www.youtube.com/watch?v=dSwDZmvtGZY

Example: A* for self-driving car

- https://www.youtube.com/watch?v=qXZt-B7iUyw

Optimality (2): Tree vs. Graph Search

- **Tree search:**
  - A* is optimal if heuristic is **admissible**
  - UCS is a special case (h = 0)
- **Graph search:**
  - A* optimal if heuristic is **consistent**
  - UCS optimal (h = 0 is consistent)
- Consistency implies admissibility
- In general, most natural admissible heuristics tend to be consistent, especially if from relaxed problems



A* Search: Find bug(s)

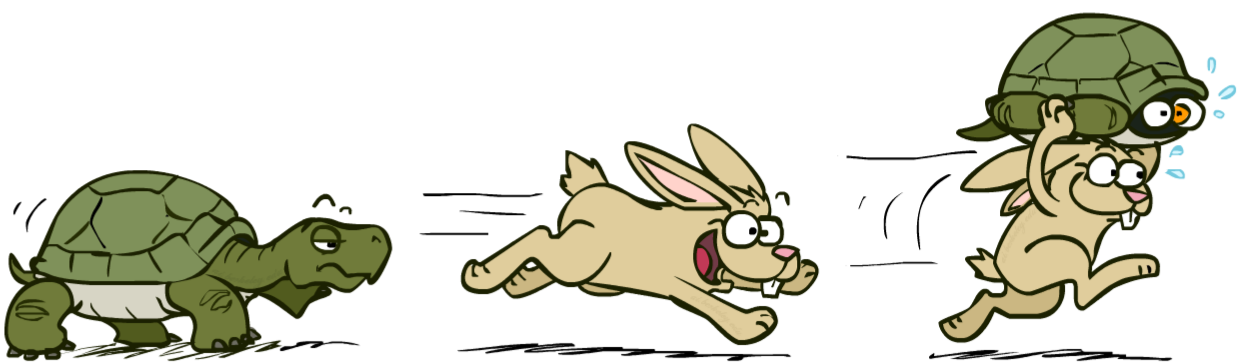
```
Node:
    state
    parent
    action_from_parent
    cost
```

```
def astar_search(problem, h=null):
        node = Node(problem.initial)
        frontier = PriorityQueue()
        frontier.append(node, null, null, 0)  //initial cost=0
        explored = set()
        while frontier:
                node = frontier.pop()
                if problem.goal_test(node.state):
                        return node

                explored.add(node.state)
                for (child, action, cost) in problem.getSuccessors(node.state):
                        if child not in explored and child not in frontier:
                                hcost = h(child.state, problem)
                                nc = new Node(child, cost+hcost)
                                frontier.append(nc, cost+hcost)
        return None
```

A*: Summary

- A* uses both backward costs and (estimates of) forward costs
- A* is optimal with admissible / consistent heuristics
- Heuristic design is key: often use relaxed problems



Project 1: Due **Friday, Feb 9**

- Read FAQ on Canvas before posting questions:
- **Questions 1-4**: if you develop a correct solution for DFS, the rest will be easy modifications
- **Run autograder** after *every* question. Until you perfectly pass all the test cases, assume your code has bugs.
- Example (incomplete!) implementations: https://github.com/aimacode/aima-python/blob/master/search.py

Tips for Project 1 (cont'd)

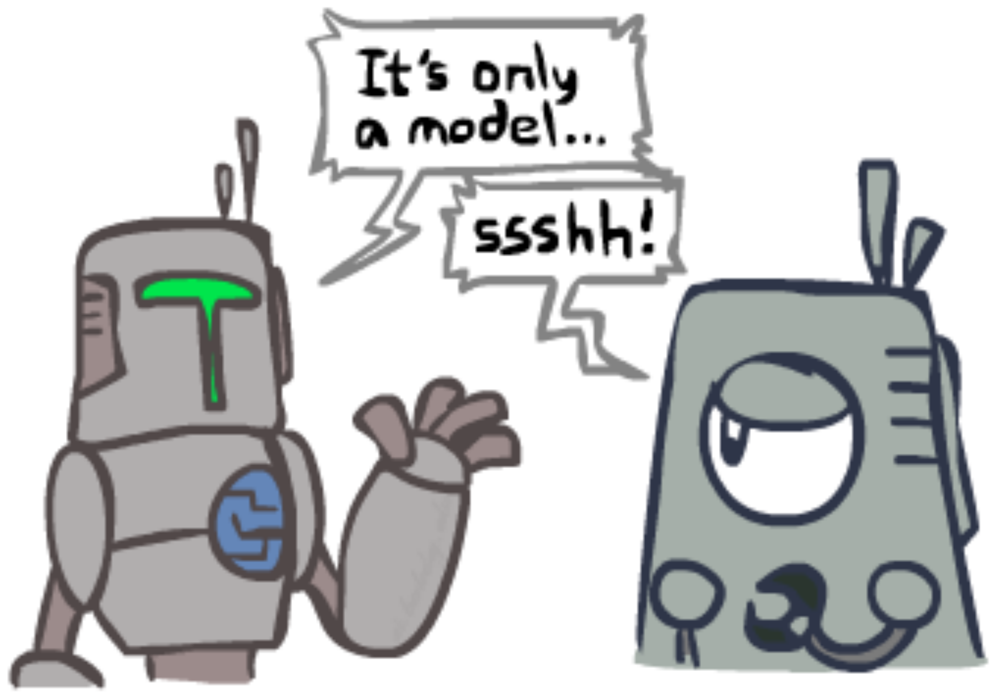
- Problems 5-8 depend on code in 1-4. Get that right (and tested) first, before moving on!
- P5/Corners problem: must visit all corners in *single* path
  - Implications for search tree, state info to update
- Heuristics for p6-8: start simple. For extra credit, think back to graph traversal algorithms from cs323.

Project 1 final hints

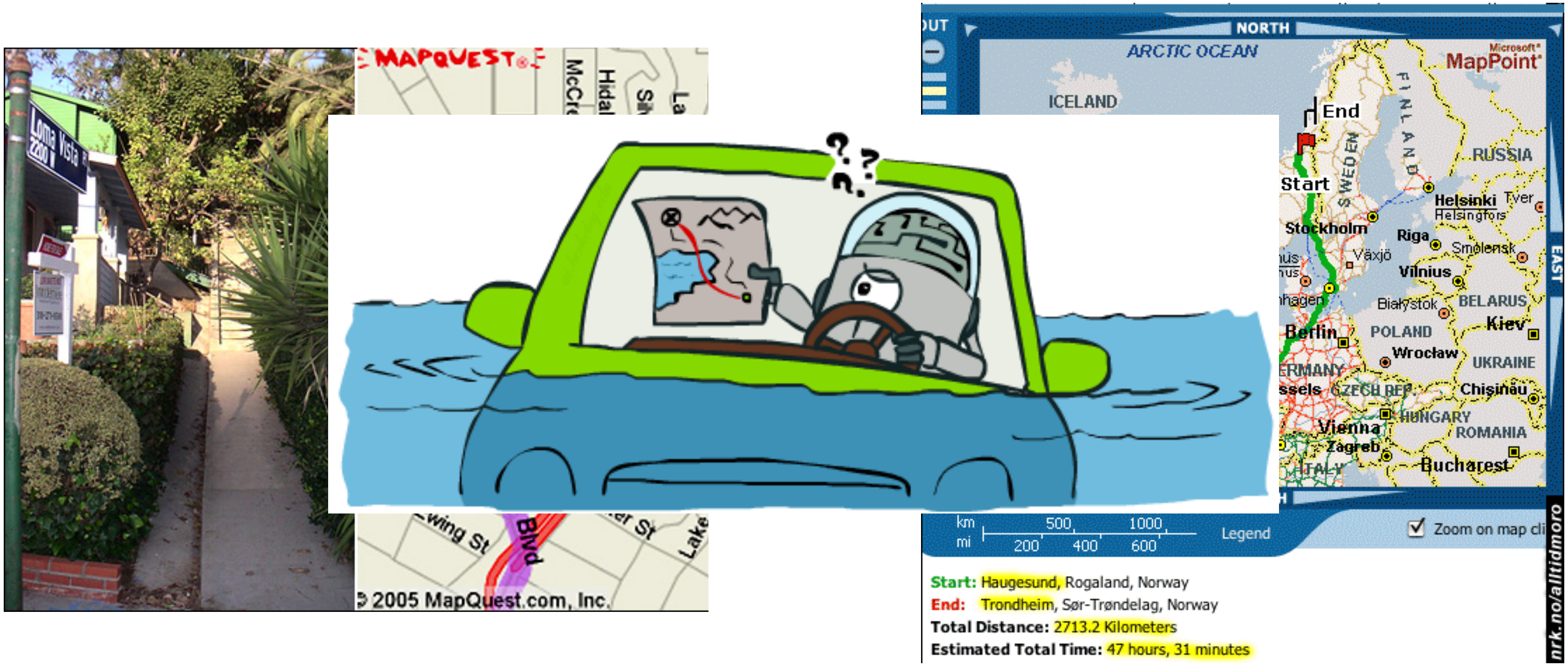
- Use Discussions, read FAQ before posting questions
- Suggestion: use Node class or similar. Easier to do Questions 5-8.
- Questions 5-8: more fun/creative. Leave enough time, start early.
- Most importantly: Don't Panic! Eat the elephant one question at a time.

Recap: Search and Models

- Search operates over models of the world
  - The agent doesn't actually try all the plans out in the real world!
  - Planning is all "in simulation"
  - Your search is only as good as your models…



Search Gone Wrong?



Properties of A* w/ consistent heuristics

- Complete?

- Time?

- Space?

- Optimal?

Properties of A* w/ consistent heuristics

- Complete? Yes (unless there are infinitely many nodes with f ≤ *f(G)* , i.e. step-cost > ε)

b^d

- Time/Space? Exponential*:

- Optimal? Yes

* Can be O(n) iff heuristic is exact, or nearly exact (ignoring heuristic computation)

Quiz

- True or False: A* can find a more optimal <u>solution</u> than UCS.

Quiz

- True or False: A* can <u>expand more nodes</u> than UCS

Quiz

- True or False: A* with consistent heuristics can expand more nodes than UCS
  - Exercise on board: proof

Properties of A* w/ consistent heuristics

- Complete? Yes (unless there are infinitely many nodes with f $\leq f(G)$, i.e. step-cost > ε)

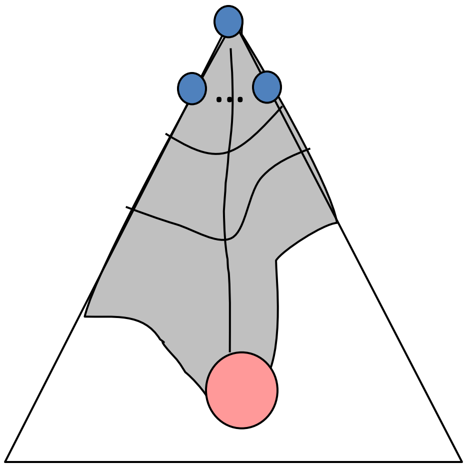
$$b^d$$

- Time/Space? **Exponential*:**
- Optimal? Yes
- *Optimally Efficient given heuristic h*: Yes*
  - (no algorithm with same heuristic is guaranteed to expand fewer nodes)

* Can be O(n) iff h(n) is exact, or nearly exact (ignoring heuristic computation)

A* is still exponential in Space

- How can we solve the **memory problem** for A* search?



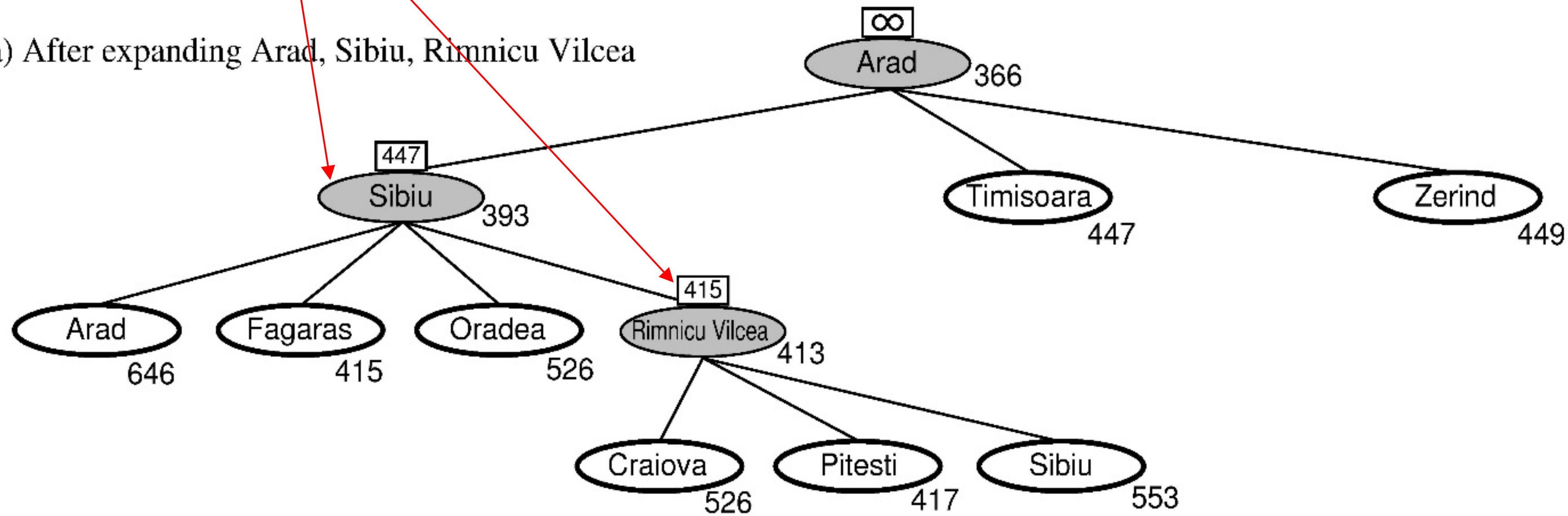
Memory Bounded Heuristic Search: **Recursive BFS**

- Idea: Try something like <u>iterative deepening DFS</u>, but let's not forget **everything** about the branches we have partially explored.
  - Run DFS up to fixed depth k, if failed, increase k
- *Remember the <u>best f-value</u> we have found so far in the branch we are deleting.*

RBFS Example (1)

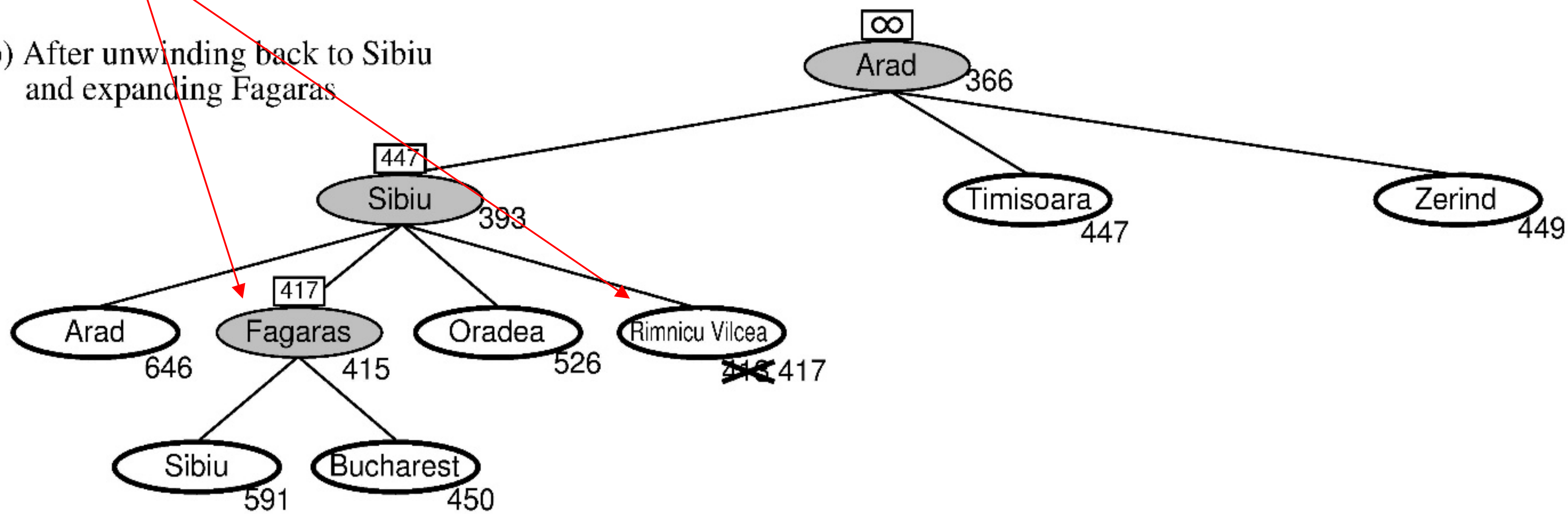
best alternative
over fringe nodes,
which are not children:
i.e. do I want to back up?

(a) After expanding Arad, Sibiu, Rimnicu Vilcea



best alternative
over fringe nodes,
which are not children:
i.e. do I want to back up?

RBFS:



(b) After unwinding back to Sibiu and expanding Fagaras

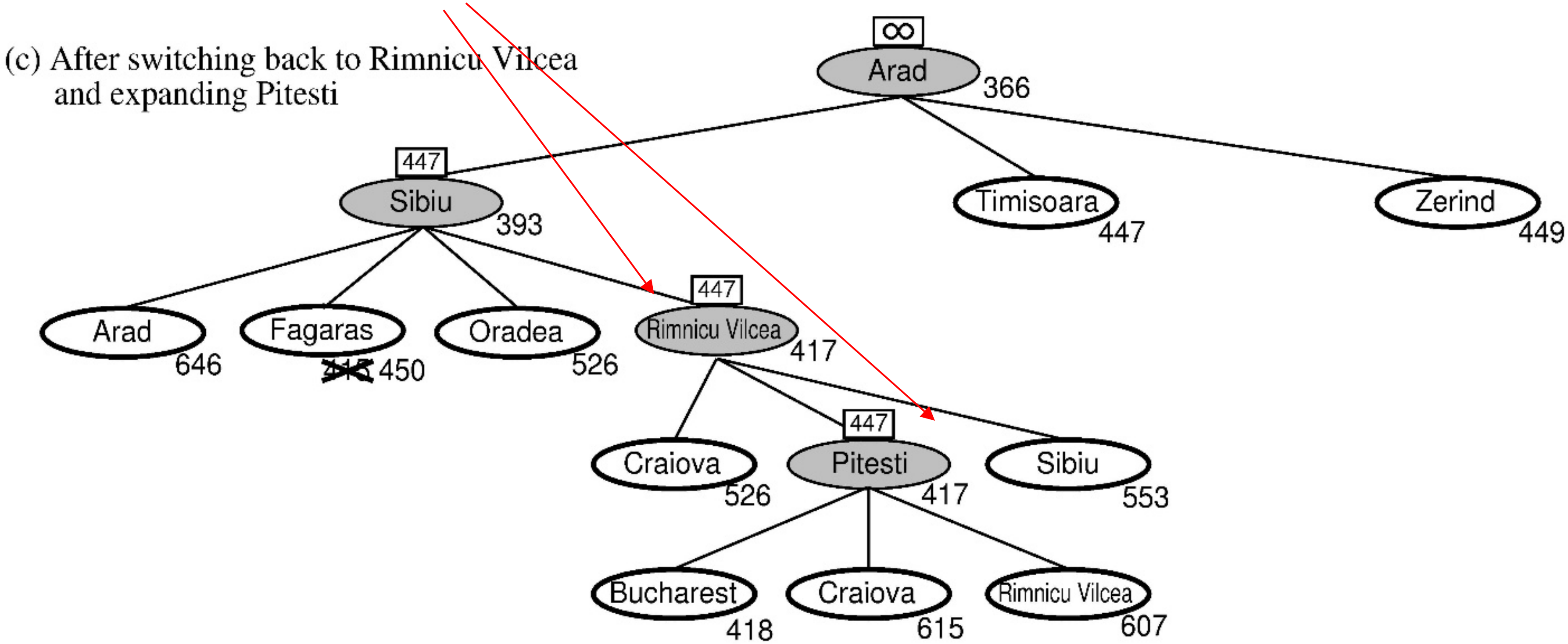
RBFS changes its mind very often in practice.

This is because the f=g+h become more accurate (less optimistic) as we approach the goal. Hence, higher level nodes have smaller f-values and will be explored first.

Problem: We should keep in memory whatever we can.

best alternative
over fringe nodes,
which are not children:
i.e. do I want to back up?

RBFS:



RBFS changes its mind very often in practice.

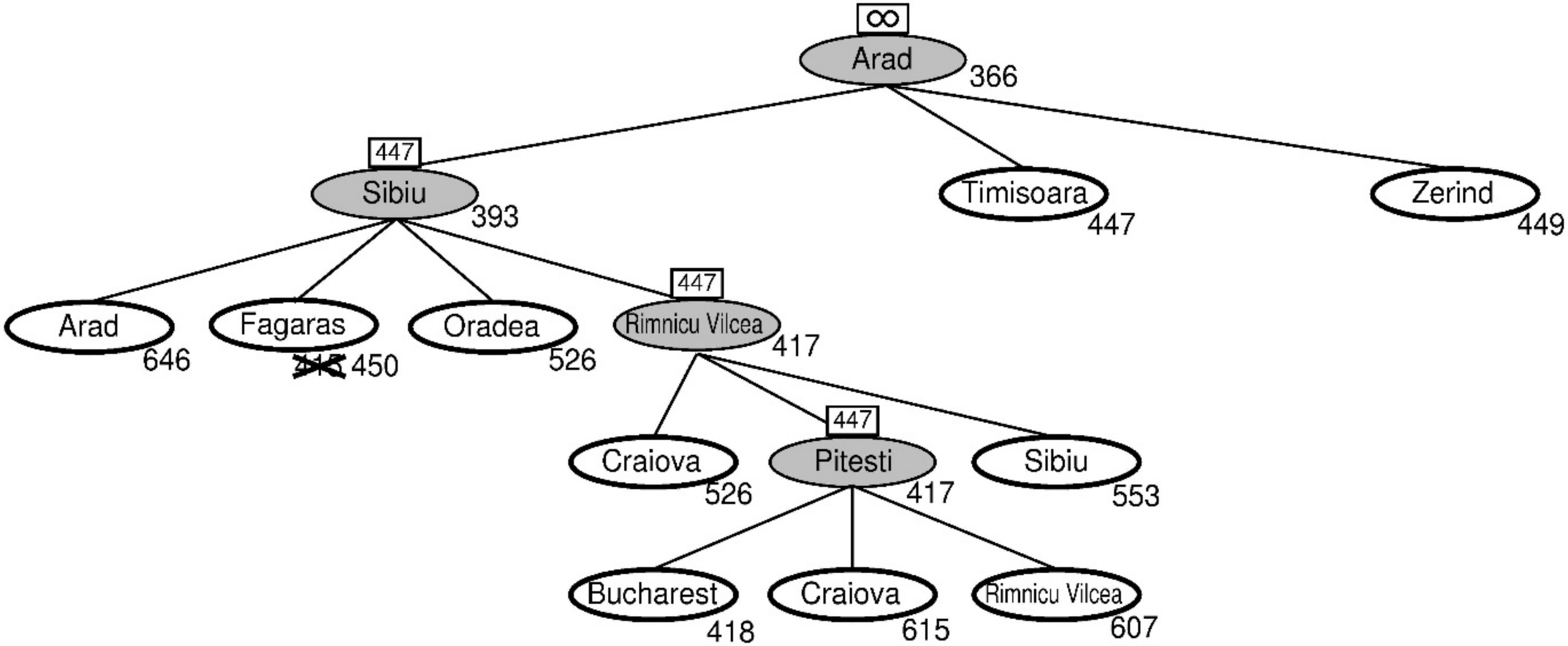
Problem: We should keep in memory whatever we can.

Simple-Memory Bounded A*

- This is like A*, but when memory is full we delete the worst node (largest f-value).
- Like RBFS, we remember the best descendent in the branch we delete.
- If there is a tie (equal f-values) we delete the oldest nodes first.
- simple-MBA* finds the optimal *reachable* solution given the memory constraint.
- Time can still be exponential.

A Solution is not reachable
if a single path from root to goal
does not fit into memory

Simple-MBA*: example



- Delete

Local search algorithms

- In many optimization problems, the path to the goal is irrelevant; the goal state itself is the solution

- State space = set of "complete" configurations
- Find configuration satisfying constraints, e.g., n-queens

- In such cases, we can use local search algorithms keep a single "current" state, try to improve it

Local Search

Where we consider more realistic search problems

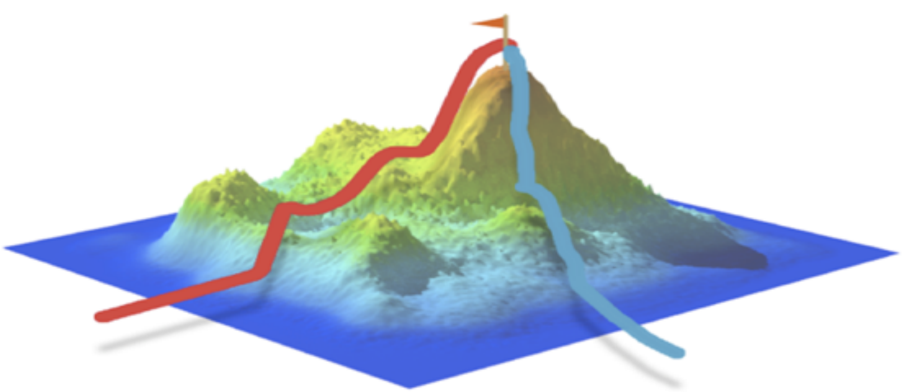
With some slides adapted from Mitch Marcus, Dan Klein

Google's Deep Mind Wins at Go

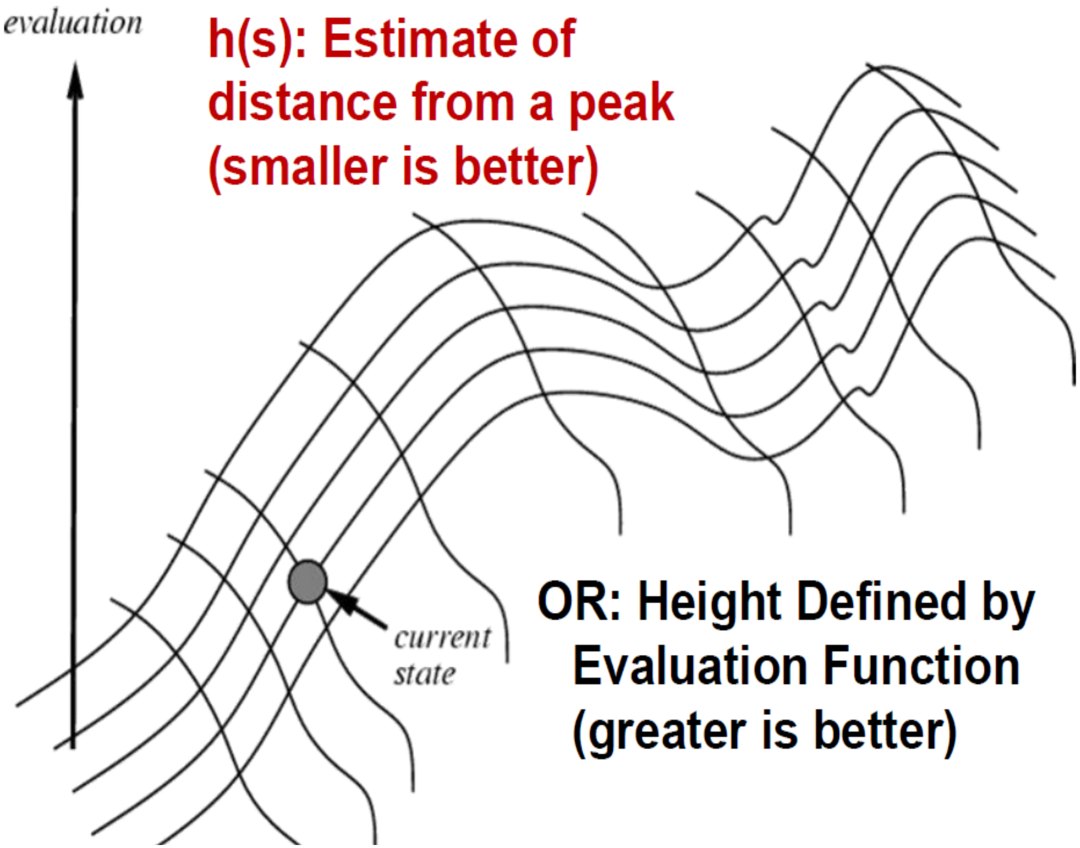
- http://deepmind.com/alpha-go.html
- "Simple" game, but difficult to master
- Orders of magnitude more states than chess (x "googol")
- Traditional search (A*) not feasible
- Google solution:
  - Local search w/ restarts (this lecture)
  - "deep learning" to estimate state values (instead of heuristics) ← observing human players
  - https://googleblog.blogspot.com/2016/01/alphago-machine-learning-game-go.html

Searching Large (or Infinite) Spaces

- Too many states to explore using A* or variants
  - Large or infinite branching factor (continuous)
- "Reasonable" solution is good enough
- Local search idea: **start with initial guess and incrementally improve**



Hill Climbing



h(s): Estimate of distance from a peak (smaller is better)

OR: Height Defined by Evaluation Function (greater is better)

Hill Climbing: Algorithm

I. **While (∃ uphill points):**
   - Move in the direction of increasing evaluation function ***f***

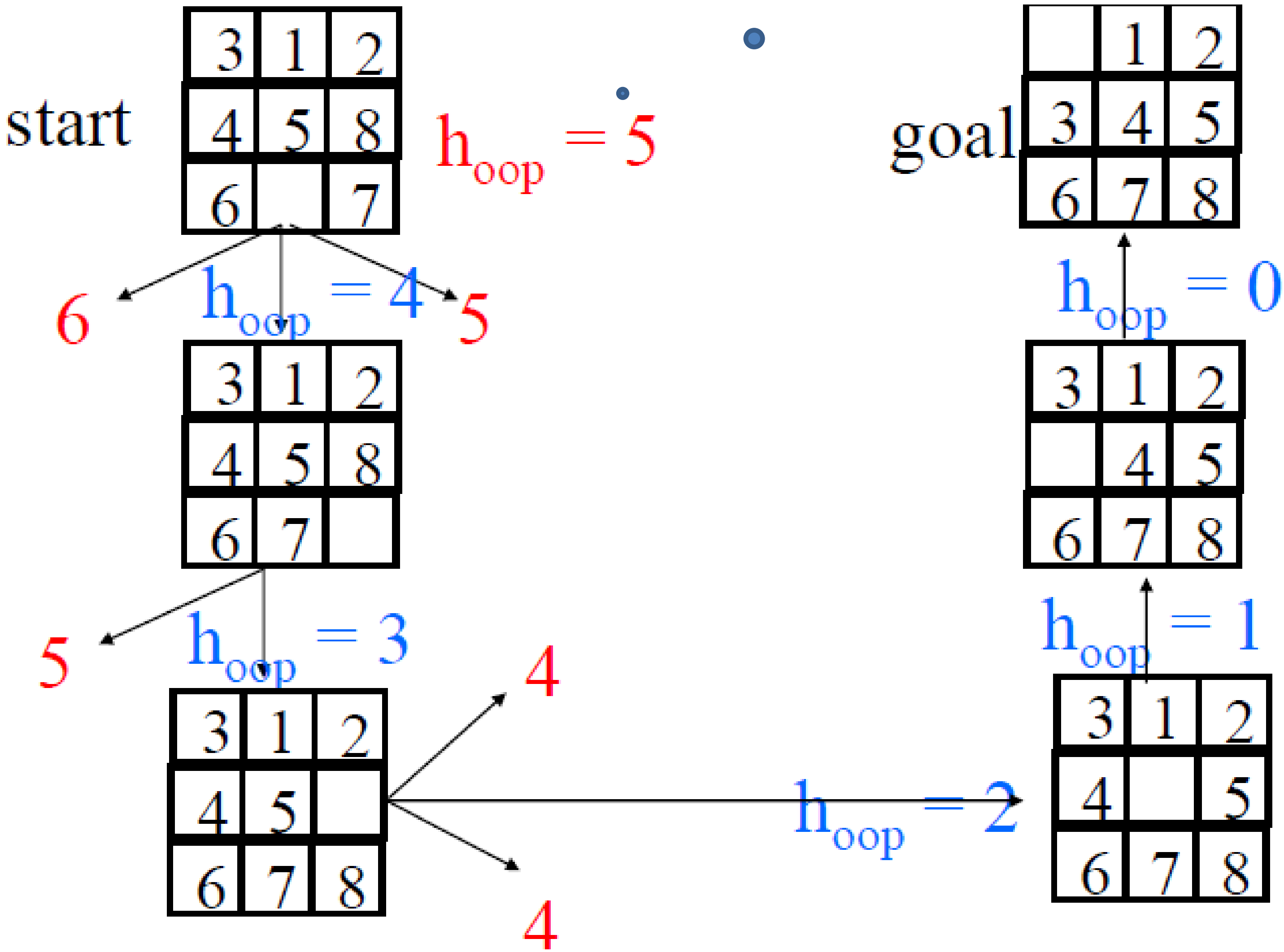
II. **Let** $s_{next} = \arg\max_{s} f(s)$ **,** ***s*** **a successor state to the current state n**
   - If $f(n) < f(s)$ then move to s
   - Otherwise halt at n

- **Properties:**
  - Terminates when a peak is reached.
  - Does not look ahead of the immediate neighbors of the current state.
  - Chooses randomly among the set of best successors, if there is more than one.
  - Doesn't *backtrack*, since it doesn't remember where it's been
- **a.k.a. *greedy local search***

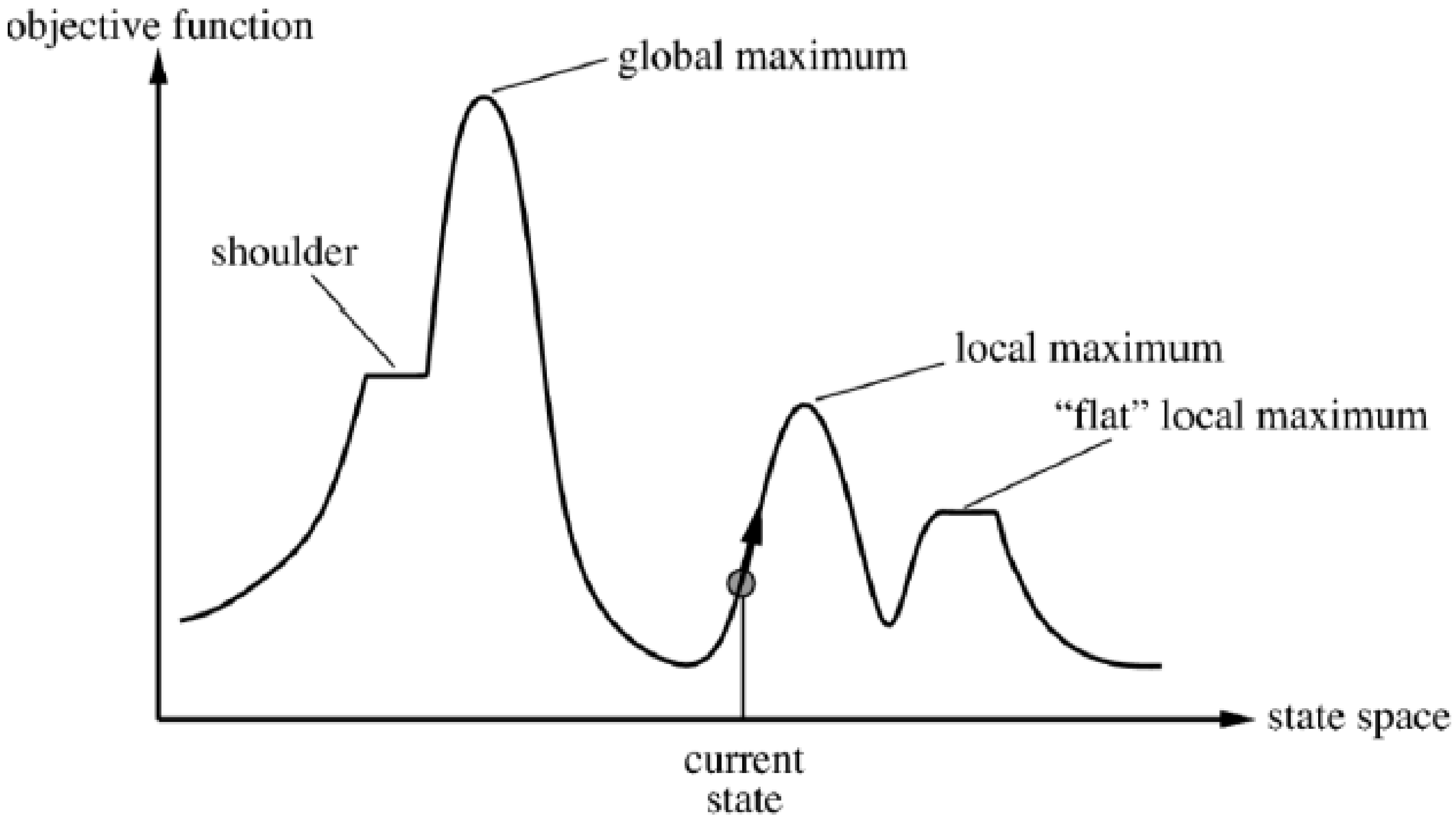
"Like climbing Everest in thick fog with amnesia"

Toy Example: Tiles

h= out of place (oop) tiles

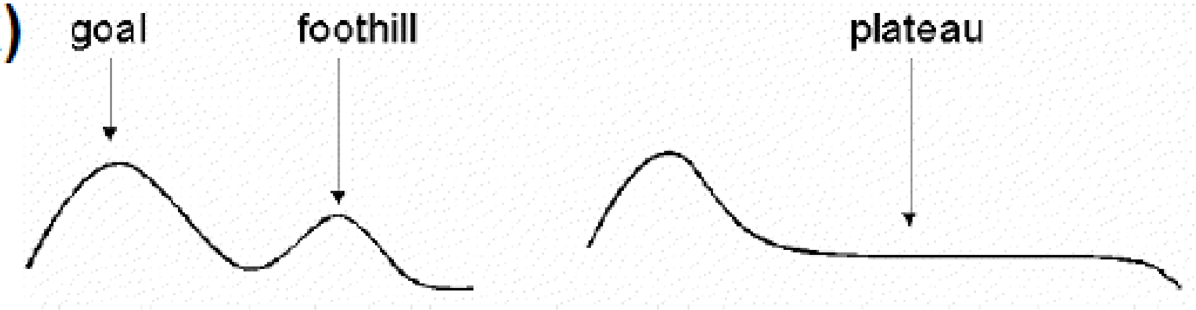


Analyzing Hill Climbing Algs

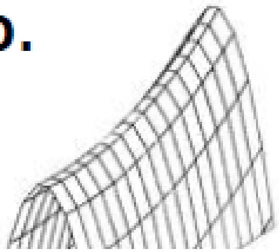


Drawbacks of Hill Climbing

- Local Maxima: peaks that aren't the highest point in the space
- Plateaus: the space has a broad flat region that gives the search algorithm no direction (random walk)

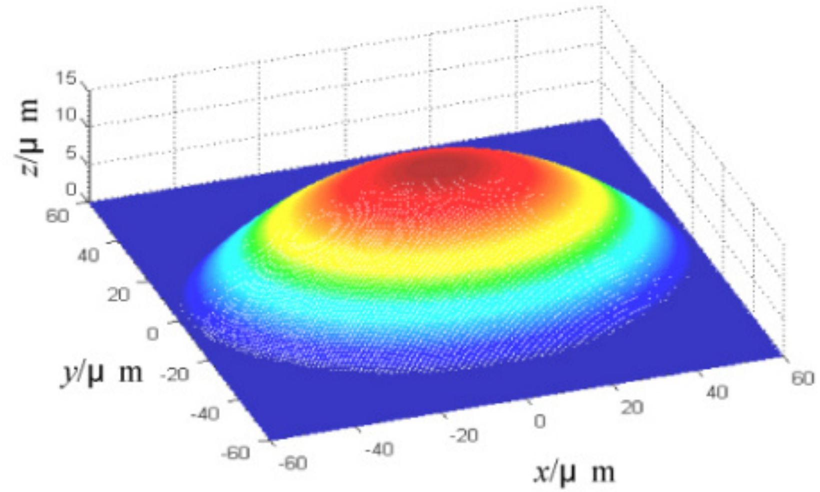


- Ridges: dropoffs to the sides; steps to the North, East, South and West may go down, but a step to the NW may go up.

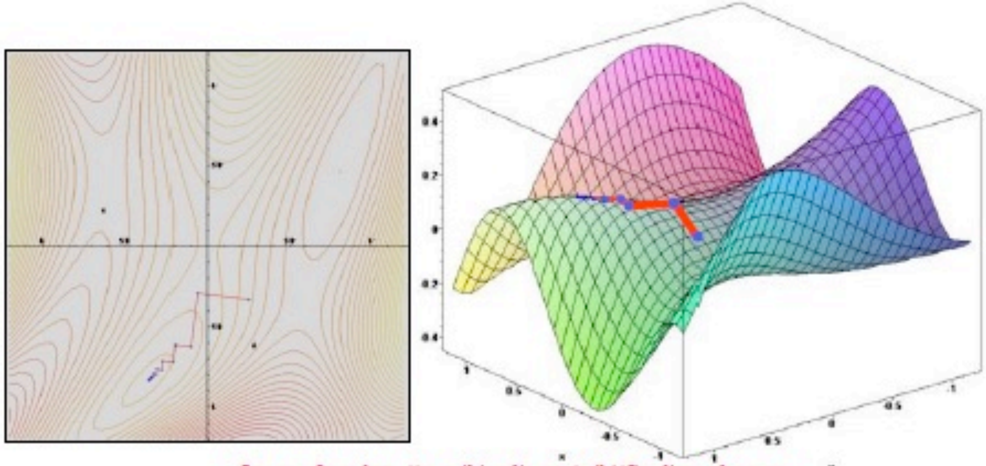


"Easy" Problems: Convex Surface

- No local maxima (only 1 peak)
- Hill climbing works great
- Can we make it faster?



Gradient Descent (Steepest Descent)



Images from http://en.wikipedia.org/wiki/Gradient_descent

- Gradient descent procedure for finding the $arg_x\ min\ \ f(x)$
  - choose initial x_0 randomly
  - repeat
    - $x_{i+1} \leftarrow x_i - \eta f'(x_i)$
  - until the sequence $x_0, x_1, \ldots, x_i, x_{i+1}$ converges
- Step size η (eta) is small (perhaps 0.1 or 0.05)

http://vis.supstat.com/2013/03/gradient-descent-algorithm-with-r/

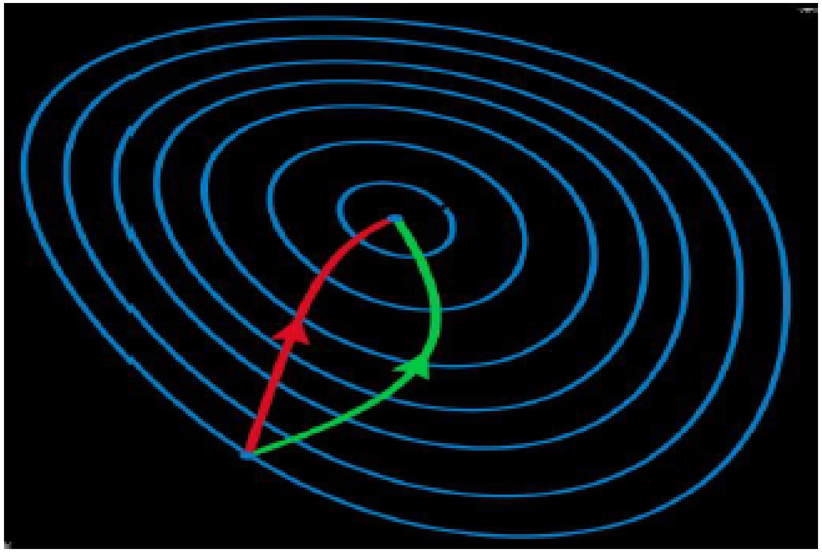
Gradient Ascent vs. Newton-Ralphston

- A reminder of Newton's method from Calculus:

  $x_{i+1} \leftarrow x_i - \eta f'(x_i) / f''(x_i)$

- Newton,s method uses 2nd order information (the second derivative, or, curvature) to take a more direct route to the minimum.

- The second-order information is more expensive to compute, but converges quicker.



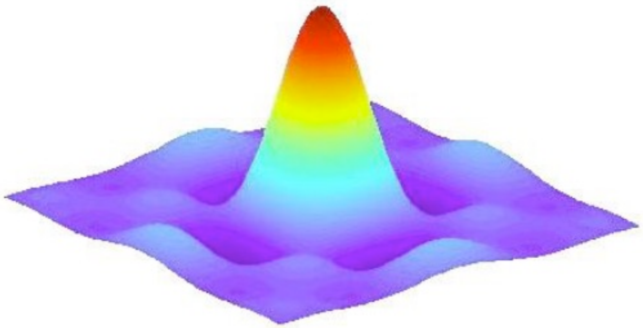
Contour lines of a function
Gradient descent (green)
Newton,s method (red)

Image from http://en.wikipedia.org/wiki/Newton's_method_in_optimization

(this and previous slide from Eric Eaton)

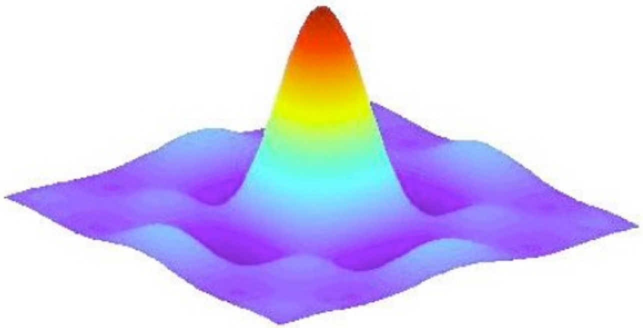
Problem: Non-Convex Surfaces

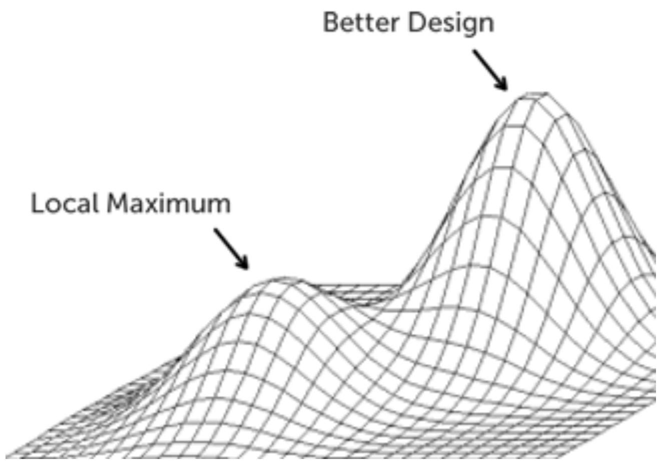
- Realistic problems:
  - Many local suboptimal maxima
  - Easy to get trapped

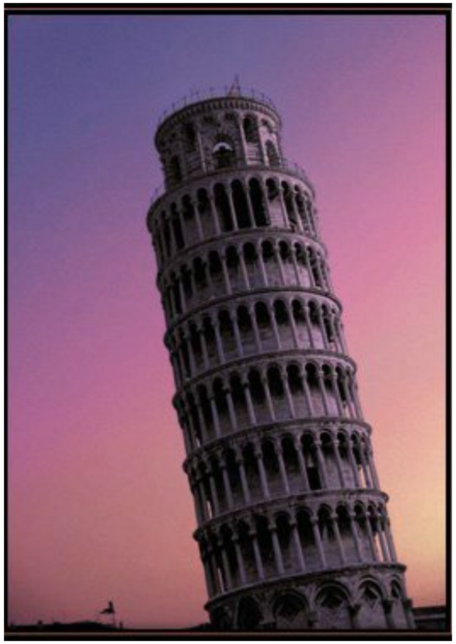


Problem: Non-Convex Surfaces

- Realistic problems:
  - Many local suboptimal maxima
  - Easy to get trapped
- Examples:







Solving the Problems

- Allow backtracking (What happens to complexity?)
- Stochastic hill climbing: choose at random from uphill moves, using steepness for a probability
- Random restarts: “If at first you don’t succeed, try, try again.”
- Several moves in each of several directions, then test
- Jump to a different part of the search space

Random Restart (Monte-Carlo methods)

- Idea: restart hill climbing algorithm from random start configurations
- Repeat N times.
- If reasonable sampling of space, w high prob will find global max

Monte Carlo Descent

1) S ← initial state
2) Repeat k times:
   a) If GOAL?(S) then return S
   b) S' ← successor of S picked at random
   c) if h(S') ≤ h(S) then S ← S'
   d) else
      - Dh = h(S')-h(S)
      - with probability ~ exp(–Dh/T), where T is called the "temperature", do: S ← S' [Metropolis criterion]
3) Return failure

Optimization: Simulated Annealing

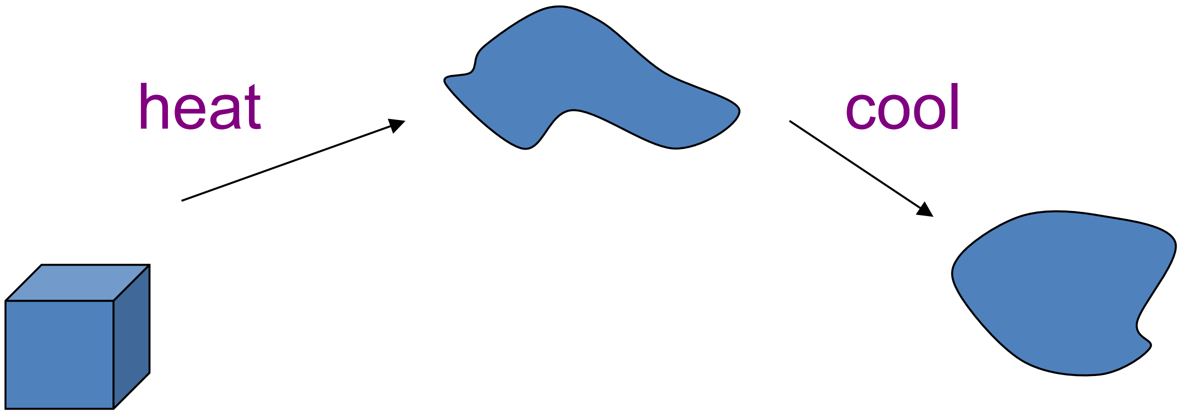
- *Annealing*: the process by which a metal cools and freezes into a minimum-energy crystalline structure (the annealing process)
- Conceptually SA exploits an analogy between annealing and the search for a minimum in a more general system.
  - AIMA: Switch viewpoint from *hill-climbing* to *gradient descent*
  - *(But: AIMA algorithm hill-climbs & larger ΔE is good…)*
- SA hill-climbing can avoid becoming trapped at local maxima.
- SA uses a random search that occasionally accepts changes that decrease objective function *f*.
- SA uses a control parameter T, which by analogy with the original application is known as the system *"temperature."*
- T starts out high and gradually decreases toward 0.

Simulated Annealing (2)

- Variant of hill climbing (maximize value)
- Tries to explore enough of the search space early on, so that the optimal solution is less sensitive to the start state
- May make some downhill moves before finding a good way to move uphill.

Simulated Annealing (3)

- Comes from the physical process of annealing in which substances are raised to high energy levels (melted) and then cooled to solid state.



- The probability of moving to a higher energy state, instead of lower is

$$p = e^{(-\Delta E/kT)}$$

where ΔE is the positive change in energy level, T is the temperature, and k is Bolzmann's constant.

Simulated Annealing (4)

- At the beginning, the temperature is high.
- As the temperature becomes lower
  - kT becomes lower
  - $\Delta E/kT$ gets bigger
  - $(-\Delta E/kT)$ gets smaller
  - e^$(-\Delta E/kT)$ gets smaller
- As the process continues, the probability of a downhill move gets smaller and smaller.
- ΔE is the change in the value of the objective function.
- Need an annealing schedule, which is a sequence of values of T: T_0, T_1, T_2, ...

Simulated Annealing Algorithm

- *current* ← start node;

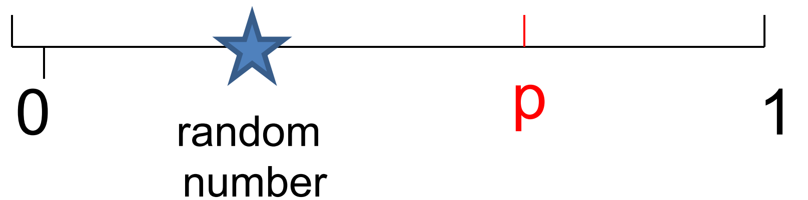
- for each T on the schedule /* need a schedule */

  - *next* ← randomly selected successor of *current*
  - evaluate next; it it's a goal, return it
  - ΔE ← *next*.Value – *current*.Value /* already negated */
  - if ΔE > 0
    - then *current* ← *next* /* better than current */
    - else *current* ← *next* with probability e^(ΔE/T)

How to select next state?

Pattern: Probabilistic Selection

- Select *next* with probability p



- Generate a random number
- If it's <= p, select *next*

Simulated Annealing Properties

- At a fixed "temperature" T, state occupation probability reaches the Boltzman distribution: https://en.wikipedia.org/wiki/Boltzmann_distribution

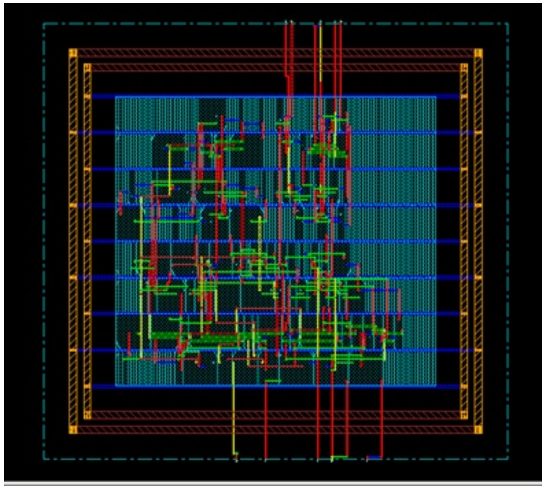
$$p_i = \frac{e^{-\varepsilon_i/kT}}{\sum_{j=1}^{M} e^{-\varepsilon_j/kT}}$$

- If T is decreased **slowly enough** (very slowly), the procedure will reach the best state.
- **Slowly enough** has proven too slow for some researchers who have developed alternate schedules.

Simulated Annealing Applications

- Basic Problems
  - Traveling salesman
  - Graph partitioning
  - Matching problems
  - Graph coloring
  - Scheduling
- Engineering
  - VLSI design
    - Placement
    - Routing
    - Array logic minimization
    - Layout
  - Facilities layout
  - Image processing
  - Code design in information theory
  - Chemistry: molecular structure:

http://toddwschneider.com/posts/traveling-salesman-with-simulated-annealing-r-and-shiny/



http://www.sciencedirect.com/science/article/pii/S0166128097001954

Local Beam Search

- Keeps more previous states in memory
  - Simulated annealing just kept one previous state in memory.
  - This search keeps k states in memory.

    - randomly generate k initial states
    - if any state is a goal, terminate
    - else, generate all successors and select best k
    - repeat

Quick Review/Quiz

- What is a difference between a game state and search node?
- What do we lose if our A* heuristic not admissible?
- Why can’t we use A* for “large” problems?

When to Use Search Techniques?

1) The search space is small, and
   - No other technique is available, or
   - Developing a more efficient technique is not worth the effort

2) The search space is large, and
   - No other available technique is available, and
   - There exist “good” heuristics

Announcements

- Reminder: project 1 due **This Friday 2/9 at 8 pm**
- Submit on Canvas
- Discuss on Canvas, but do not post or share code -- will be checked for plagiarism
- **No lecture on Thursday – instead, extra TA help session with Project 1**
- Make-up lecture: TBA