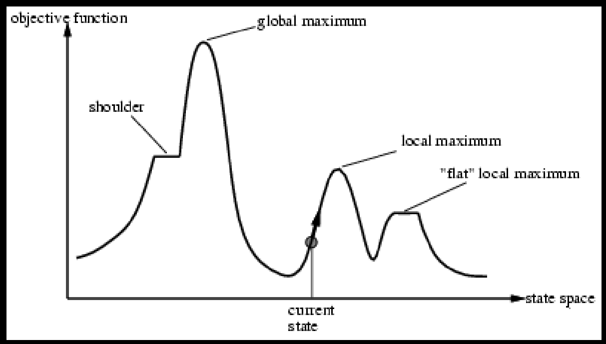
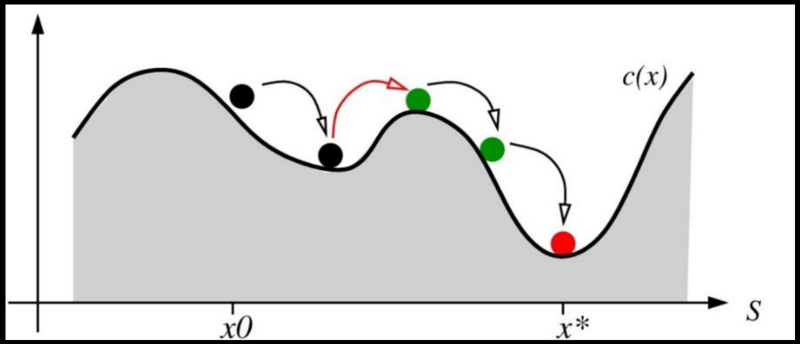
Searching for a Path

Last time…

- We saw how to frame optimization as a graph search problem





- But what if we need *a path* from a start to a goal that is feasible/optimal?
- Motion planning for robots requires solving this problem!

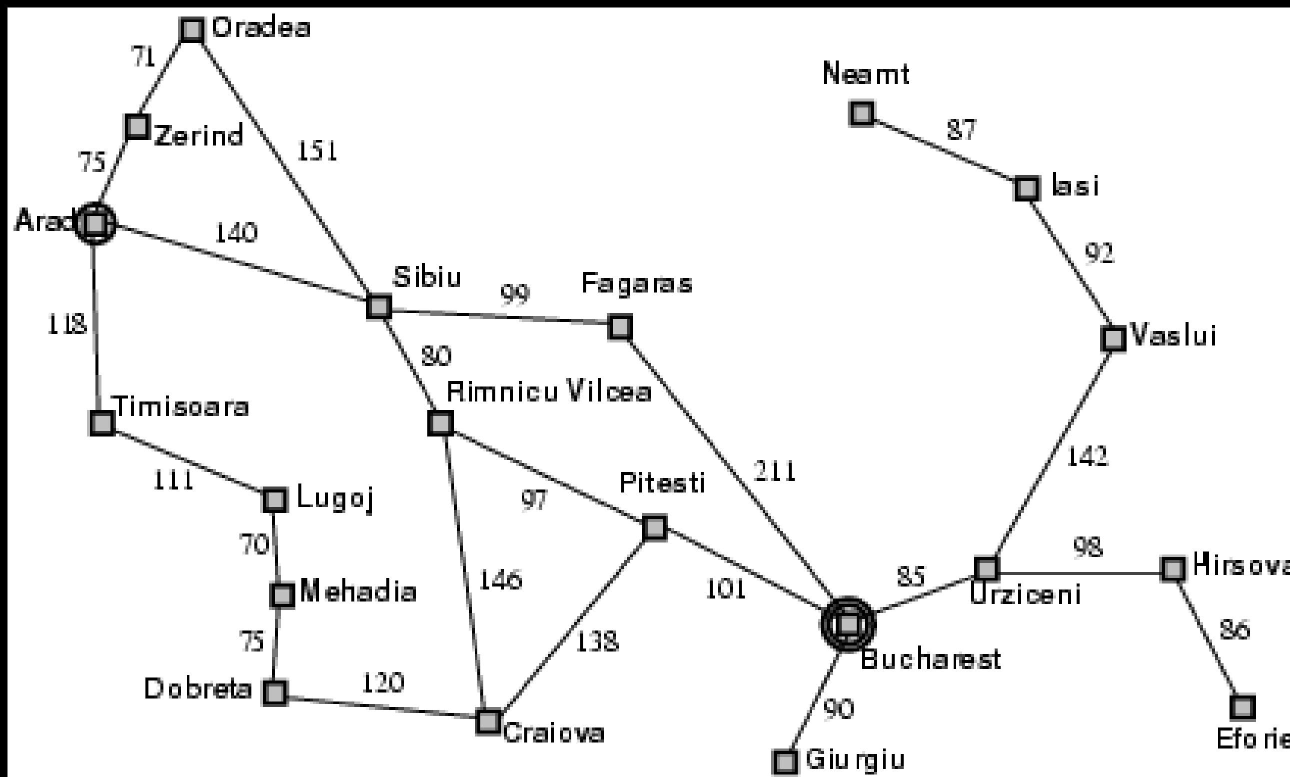
Outline

- Formulating a path search problem
  - Examples
  - General considerations for robotics
- Tree search algorithms
  - Uninformed search
  - Informed search

Formulating a path search problem

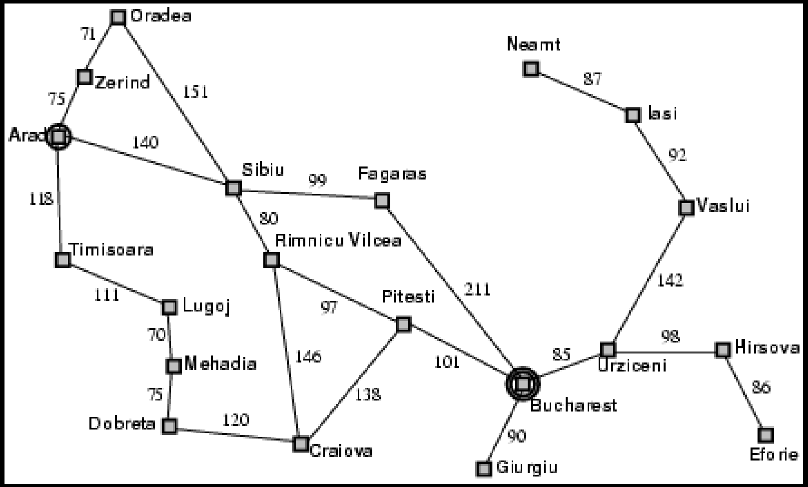
1. State Space
2. Successor Function
3. Actions
4. Action Cost
5. Goal Test

Example: Romanian roadmap

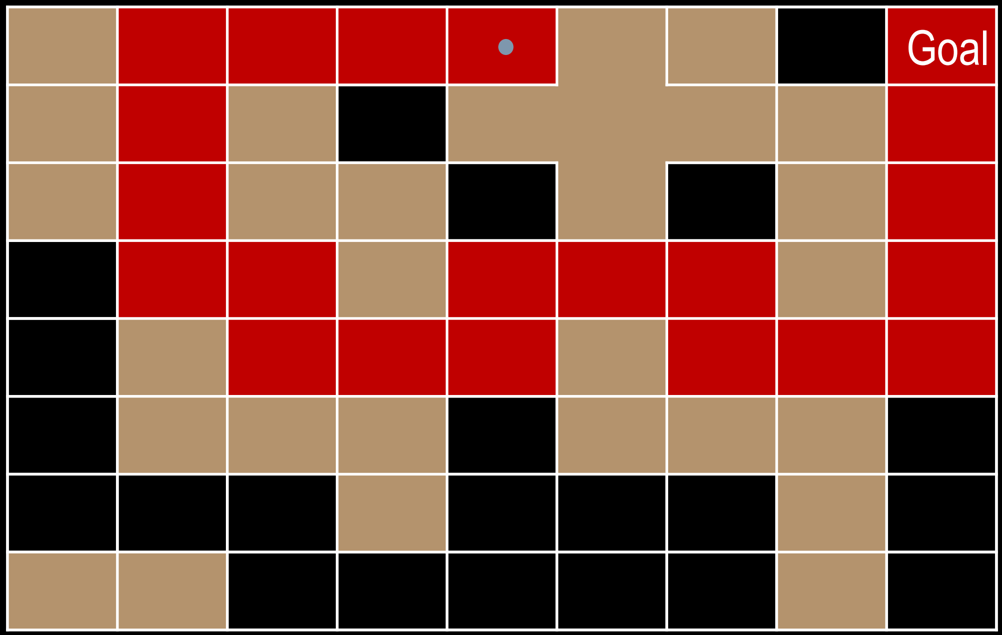


Example: A Romanian roadmap

1. State Space
   - The set of cities
2. Successor Function
   - A city's successors are cities directly connected to it (its neighbors)
3. Actions
   - Move to a neighboring city
4. Action Cost
   - Distance between the cities
5. Goal Test
   - Check if at goal city



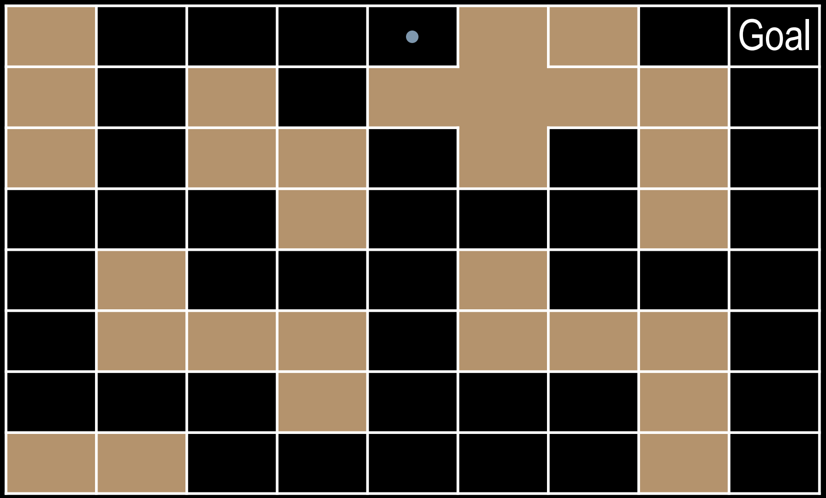
Example : Point robot in a maze:

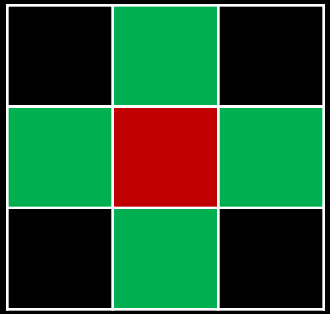


- Find a sequence of free cells that goes from start to goal

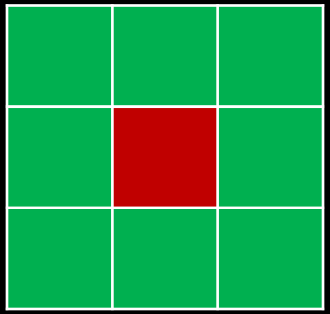
Point Robot Example

1. State Space
   - The space of cells, usually in x,y coordinates
2. Successor Function
   - A cell's successors are its free neighbors
   - 4-connected vs. 8-connected
3. Actions
   - Move to a neighboring cell
4. Action Cost
   - Distance between cells traversed
   - Are costs the same for 4 vs 8 connected?
5. Goal Test
   - Check if at goal cell
   - Multiple cells can be marked as goals





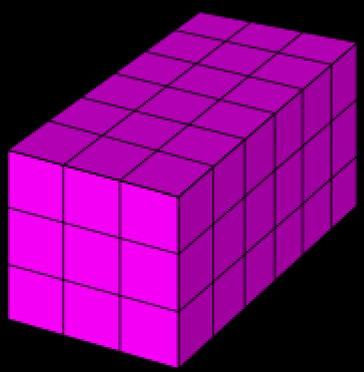
4-connected



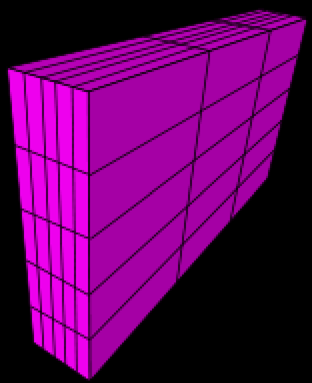
8-connected

Formulating the problem: State space

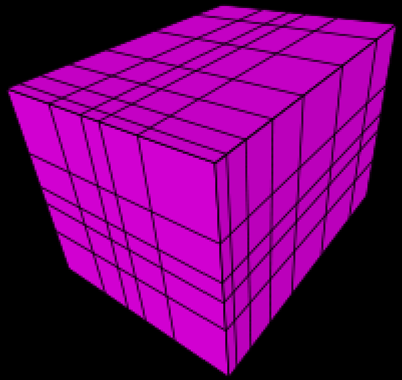
- For motion planning in robotics, state space is often a grid
- There are many kinds of grids!



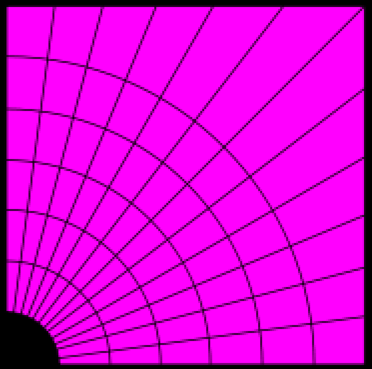
Cartesian Grid



Regular Grid



Rectilinear Grid

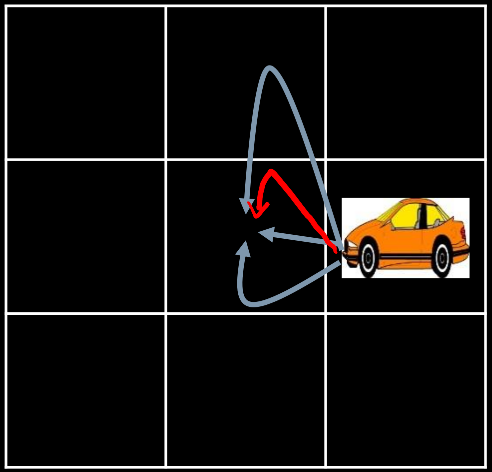


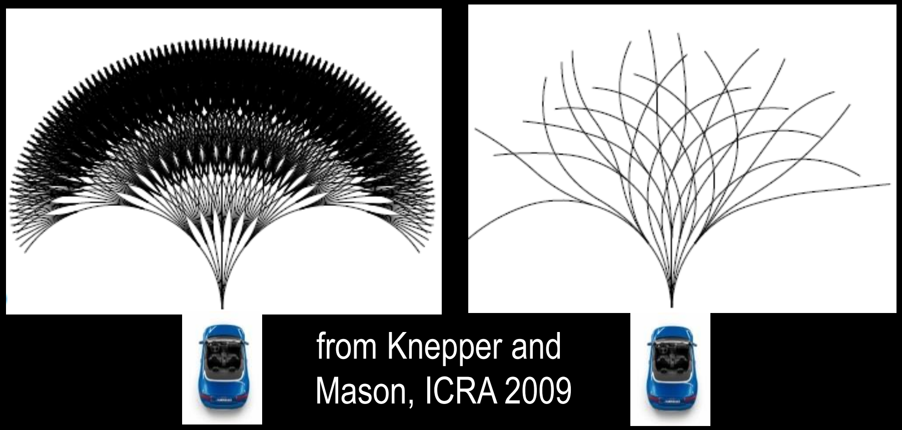
Curvilinear Grid

- The choice of grid (i.e. state space) is crucial to performance and accuracy
- Remember, the world is really continuous; these are all approximations

Formulating the problem: Actions

- Actions in motion planning are also often continuous
- For example, many ways to move between neighboring cells
- Usually pick a discrete action set *a priori*
- What are the tradeoffs in picking action sets?

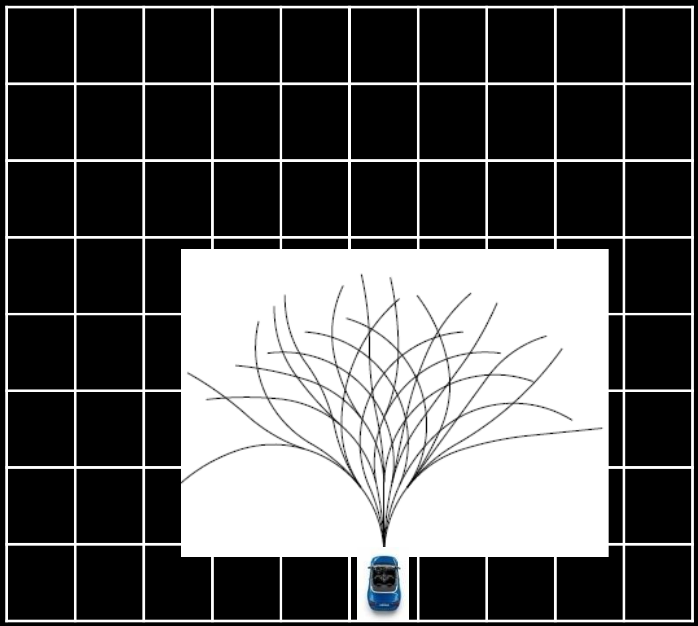




from Knepper and Mason, ICRA 2009

Formulating the problem: Successors

- These are largely determined by the action set
- Successors may not be known a priori
  - I.e. you have to try each action in your action set to see which cell you end in



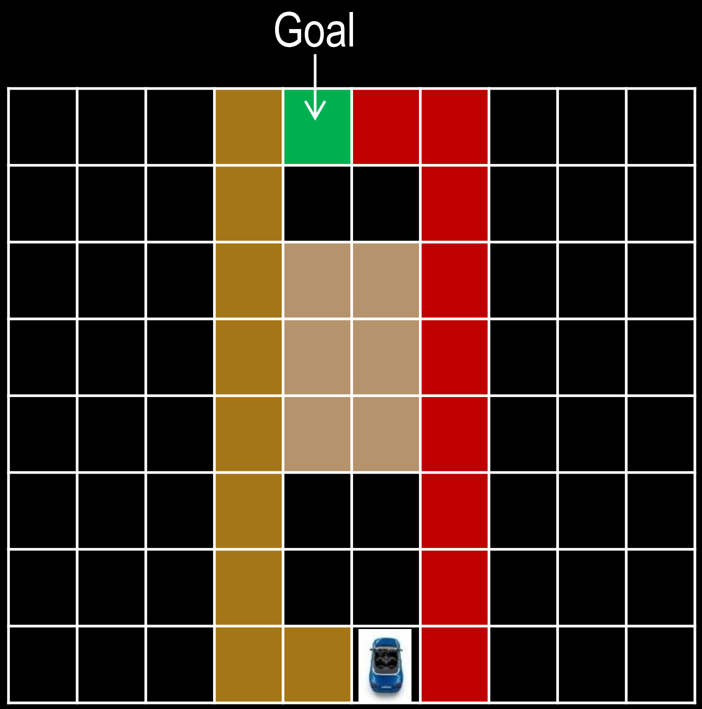
Formulating the Problem: Action Cost

- Depends on what you're trying to optimize
  - Minimum Path Length: Cost is distance traversed when executing action
  - What is action cost for path smoothness?
- Sometimes we consider more than one criterion
  - Linear combination of cost functions (most common):

    $Cost = a_1C_1 + a_2C_2 + a_3C_3 \ldots.$

Formulating the Problem: Goal Test

- Goals are most commonly specific cells you want to get to
- But they can be more abstract, too!
- Example Goals:
  - A state where X is visible
  - A state where the robot is contacting X
  - Topological goals



A topological goal example: go **right** around the obstacle (need whole path to evaluate if goal reached)

Finding a Path:
Tree Search Algorithms

Tree Search Algorithms

```
function Tree-Search(problem, strategy)
    Root of search tree <- Initial state of the problem
    While 1
        If no nodes to expand
                return failure
        Choose a node n to expand according to strategy
        If n is a goal state
                return solution path  //back-track from goal to
                                      //start in the tree to get path
        Else
                NewNodes <- expand n
                Add NewNodes as children of n in the search tree
```

Strategies are evaluated in terms of

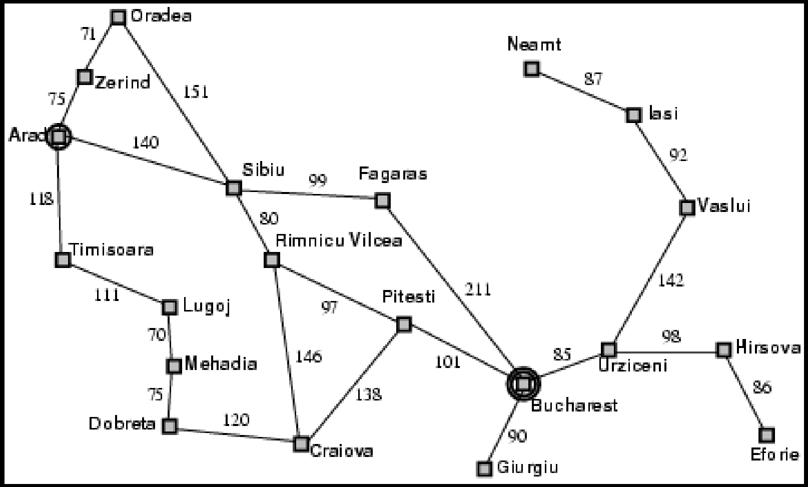
- completeness: does it always find a solution if one exists?
- optimality: does it always find a least-cost solution?

- Two types of complexity
  - time complexity: number of nodes visited
  - space complexity: maximum number of nodes in memory

Time and space complexity are measured in terms of

- *b:* maximum branching factor of the search tree (may $\rightarrow\infty$)
- *d:* depth of the least-cost solution
- *m*: maximum depth of the state space (may be ∞)

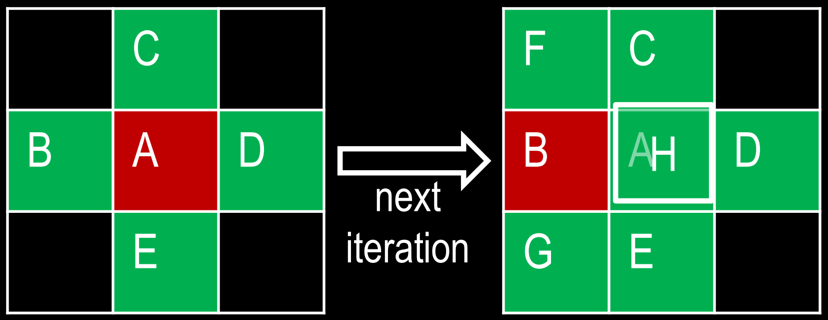
  - What is *b*? (worst case)
  - What is *d*?
  - What is *m*?



Tree Search Algorithm Implementation

- Open list: List of nodes we know about but haven't expanded yet
  - Implement as a as a queue, stack, or priority queue (depending on the algorithm)

- Need to avoid re-expanding the same state



- Solution: A *closed list* to track which nodes are already explored

Uninformed Search Strategies

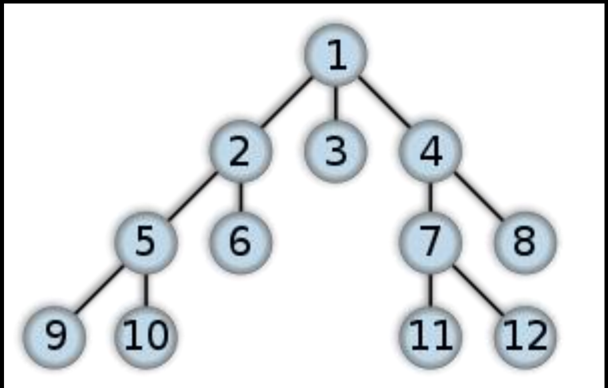
Uninformed search strategies

- Uninformed search strategies use only the information available in the problem definition

- What does it mean to be uninformed?
  - You only know which states are connected by which actions. No additional information.
  - Later we'll talk about informed search, in which you can *estimate* which actions are likely to be better than others.

Breadth-first Search (BFS)

- Main idea: build search tree in layers
- Assumes all actions have equal cost
- Open list is a queue (e.g. `std::queue`), new nodes are inserted at the **back**



- Result: "Oldest" nodes are expanded first
- BFS finds the shortest path to the goal
- When would this strategy be very inefficient?

Properties of breadth-first search

- Complete?
  - Yes (if b is finite)

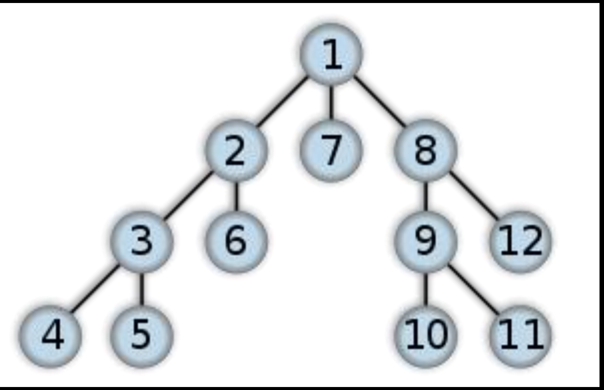
- Optimal?
  - Yes (**if cost = 1 per step**)

- Time?
  - $1+b+b^2+b^3+\ldots+b^d$ = O(b^d)

- Space?
  - $O(b^d)$ (keeps every node in memory)

Depth-first Search (DFS)

- Main idea: Go as deep as possible as fast as possible
- Assumes all actions have equal cost
- Open list is a stack, new nodes are inserted at the **front**
  - Use `std::stack`
    - Like a `std::queue`, except you can only add to the *front* of it, using `push()`
  - `std::stack` has nothing to do with the stack used in memory (i.e. stack vs. heap)



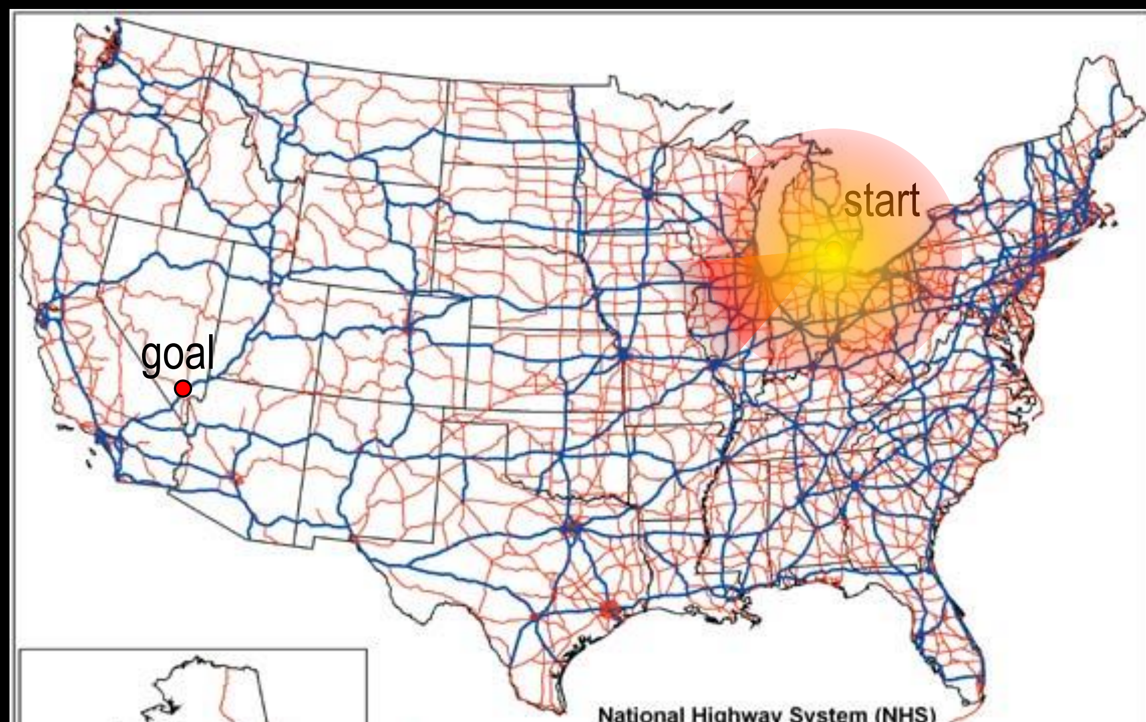
- Result: "Newest" nodes are expanded first
- DFS does NOT necessarily find the shortest path to the goal
- When would this strategy be very inefficient?

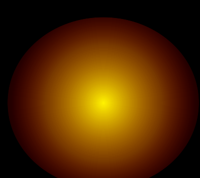
Properties of depth-first search

- Complete?
  - No: fails in infinite-depth spaces
  - Yes: in finite spaces
- Optimal?
  - No
- Time?
  - $O(b^m)$: (m is max depth of state space)
  - terrible if *m* is much larger than *d*
  - but if solutions are plentiful, may be much faster than breadth-first
- Space?
  - $O(bm)$

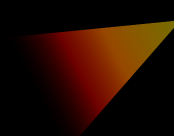
Informed Search Strategies

Search types





Uninformed search



Informed search

We'll need a new tool: `std::priority_queue`

- Similar to `std::queue`, but instead of the order being in order of insertion (last-in-first-out), in `std::priority_queue`, elements are sorted by their *priority*
- For simple types, it uses "<" to sort:

```
std::priority_queue<int> pq_int;

pq_int.push(1);
pq_int.push(7);
pq_int.push(5);
pq_int.push(3);
pq_int.push(285);

while (!pq_int.empty()) {
    int p = pq_int.top();
    pq_int.pop();
    cout << p << " ";
}
//prints 285 7 5 3 1
```

This is how you get the first element

Same as for std::queue

- Note that it re-sorts every time you push an element!

We'll need a new tool: `std::priority_queue`

- For more complex types, need to define "<", e.g:

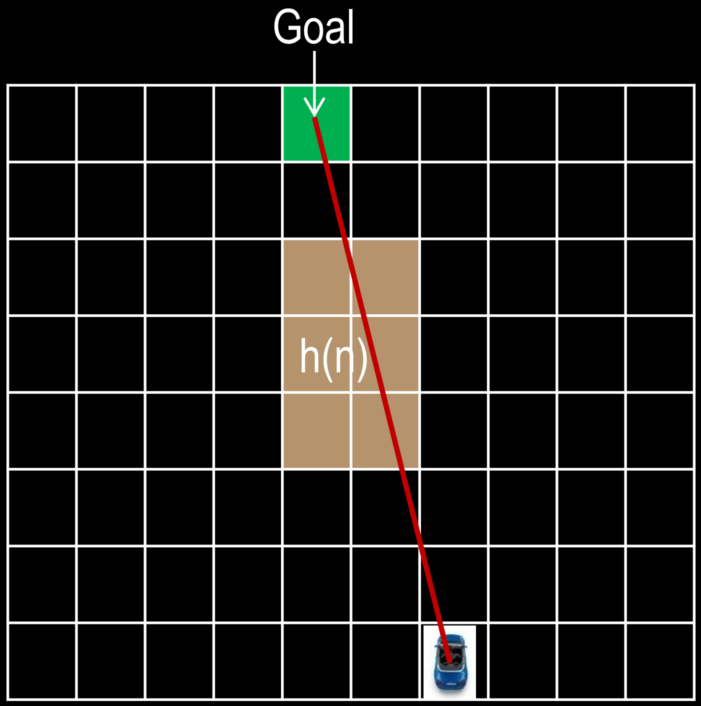
```
class Node {
public:
    int priority; //bad style: these variables should be private
    string name;
    Node(string name, int priority) : name(name), priority(priority){}
};

bool operator<(const Node& p1, const Node& p2)
{
    return p1.priority < p2.priority;
}

int main()
{
    priority_queue<Node> pq;
    pq.push(Node("N1",21));
    pq.push(Node("N2",19));
    pq.push(Node("N3",47));
    pq.push(Node("N4",23));

    while (!pq.empty()) {
        Node n = pq.top();
        pq.pop();
        cout << n.name << " " << n.priority << "\n";
    }
    return 0;
}
```

Output:

```
N3 47
N4 23
N1 21
N2 19
```

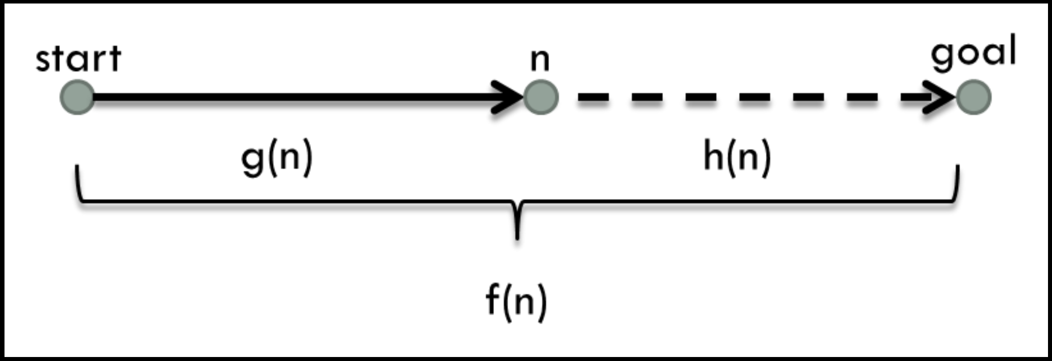
Best-first Search

- Main idea: Use Heuristic function h(n) to estimate each node's distance to goal, expand node with minimum h(n)
- Open list is a *priority queue*, nodes are sorted according to h(n) (ascending)
- Result: Works great if heuristic is a good estimate
- Does not necessarily find least-cost path
- When would this strategy be inefficient?



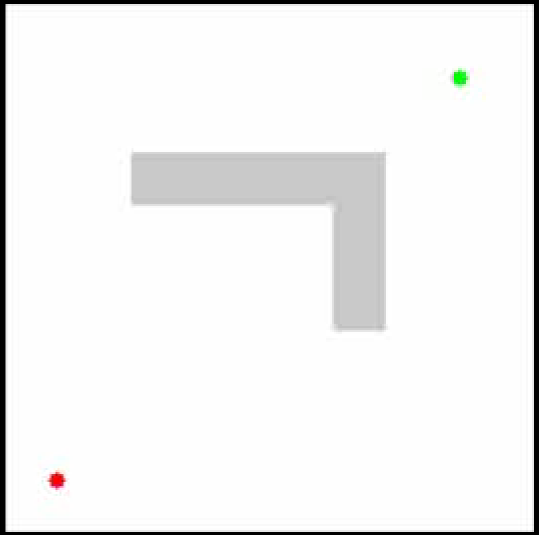
A * Evaluation functions

- Main idea: Select nodes based on estimated path to goal
- Evaluation function *f(n)* = *g(n)* + *h(n)*
  - *g(n)* = cost so far to reach *n* (cost-to-come)
  - *h(n)* = estimated cost from *n* to goal (cost-to-go)
  - *f(n)* = **estimated total cost** of path through *n* to goal



A* Search

- Open list is a *priority queue*, nodes are sorted according to f(n) (ascending)
- g(n) is sum of edge costs from root node to n



A* Search

- IMPORTANT RESULT: If $h(n)$ is *admissible*, A* will find the least-cost path!

- Admissibility: $h(n)$ must *never overestimate* the true cost to reach the goal from n

  $h(n) \leq h^*(n)$, where $h^*(n)$ is the true cost to reach goal from n

  $h(n) \geq 0$ (so $h(G) = 0$ for goals G)

- "Inflating" the hueristic may give you faster search, but least-cost path is not guaranteed

Properties of A* (w/ admissible heuristic)

- Complete?
  - Yes (unless there are infinitely many nodes with f ≤ f(G))
- Optimal?
  - Yes
- Time?
  - Exponential, approximately $O(b^d)$ in the worst case
- Space?
  - $O(b^m)$ Keeps all nodes in memory

Heuristic Consistency

- A heuristic is consistent (also known as monotonic) when it obeys

$$h(n_1) \leq C(n_1, n_2) + h(n_2)$$

$$h(G) = 0 \text{ for goals } G$$

for adjacent n_1, n_2 and with edge cost $C(n_1, n_2)$

- Ensures that it is impossible to decrease *f* by extending a path to include a neighboring node
- A consistent heuristic is always admissible

- If a heuristic is consistent, we can use a closed list
- If not consistent, we cannot use a closed list and guarantee optimality

Summary

- A search problem is defined by
  1. State Space
  2. Successor Function
  3. Actions
  4. Action Cost
  5. Goal Test

- Search types:
  - Uninformed: Uses only problem definition, no additional information
  - Informed: Relies on a heuristic to estimate cost to reach the goal

- A* is a strong approach
  - My default method for search problems

Homework

- Homework 6