

# Introduction to Multi-Agent Programming

## **4. Search Algorithms and Path-finding**

---

Uninformed & informed search, Online search, Robot Path Planning

*Alexander Kleiner, Bernhard Nebel*

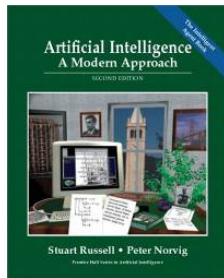
# Contents

---

- Problem-Solving Agents
- General Search (Uninformed search)
- Best-First Search (Informed search)
  - Greedy Search & A\*
- Online Search
  - Real-Time Adaptive A\*
- Robot Path Planning
  - Sampling Based Planning

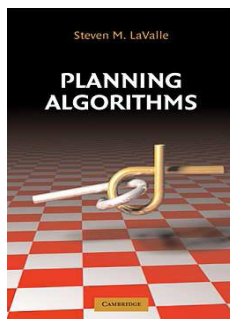
# Literature

Illustrations and content presented in this lecture were taken from:



*Artificial Intelligence – A Modern Approach, 2<sup>nd</sup> Edition*

by Stuart Russell - Peter Norvig



*Planning Algorithms*

By Steven M. LaValle

Available for downloading at: <http://planning.cs.uiuc.edu/>

# Problem-Solving Agents

→ Goal-based agents

Formulation: *goal* and *problem*

Given: *initial state*

Task: To reach the specified goal (a state) through the *execution of appropriate actions*.

→ Search for a suitable *action sequence* and execute the actions

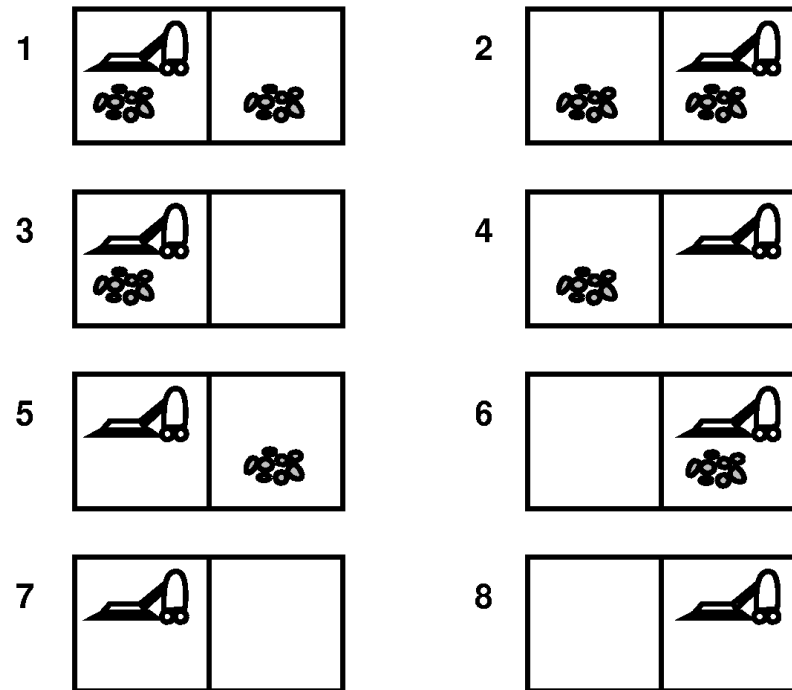
# Problem Formulation

- **Goal** formulation  
World states with certain properties
- Definition of the **state space**  
important: only the relevant aspects → abstraction
- Definition of the **actions** that can change the world state
- Determination of the **search cost** (search costs, offline costs) and the execution costs (path costs, online costs)

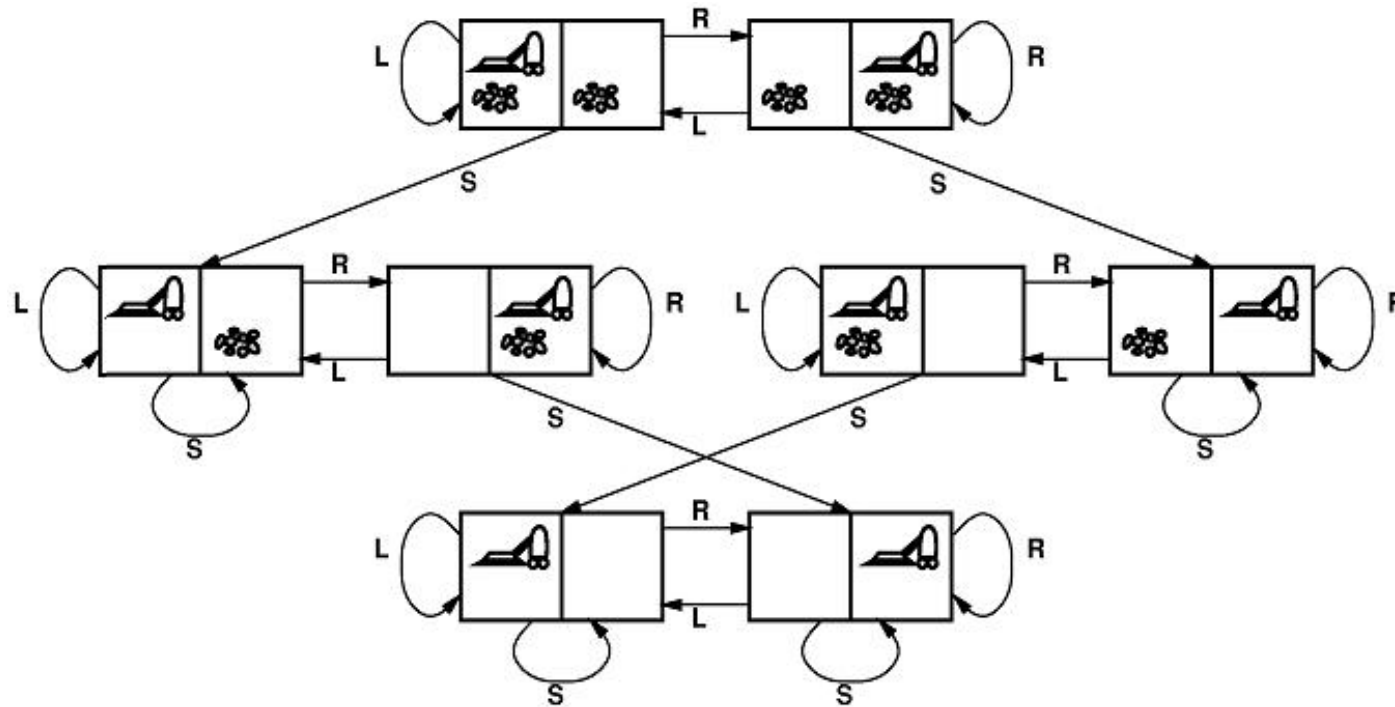
**Note:** The type of problem formulation can have a big influence on the difficulty of finding a solution.

# Problem Formulation for the Vacuum Cleaner World

- World state space:  
2 positions, dirt or no dirt  
→ 8 world states
- Successor function  
(Actions):  
Left (L), Right (R), or Suck (S)
- Goal state:  
no dirt in the rooms
- Path costs:  
one unit per action



# The Vacuum Cleaner State Space



States for the search: The world states 1-8.

# Implementing the Search Tree

## *Data structure for nodes in the search tree:*

**State:** state in the state space

**Node:** Containing a state, pointer to predecessor, depth, and path cost, action

**Depth:** number of steps along the path from the initial state

**Path Cost:** Cost of the path from the initial state to the node

**Fringe:** Memory for storing expanded nodes. For example, stack or a queue

## *General functions to implement:*

**Make-Node(state):** Creates a node from a state

**Goal-Test(state):** Returns true if state is a goal state

**Successor-Fn(state):** Implements the successor function, i.e. expands a set of new nodes given all actions applicable in the state

**Cost(state,action):** Returns the cost for executing action in state

**Insert(node, fringe):** Inserts a new node into the fringe

**Remove-First(fringe):** Returns the first node from the fringe



# General Tree-Search Procedure

**function** TREE-SEARCH(*problem*, *fringe*) **returns** a solution, or failure

*fringe*  $\leftarrow$  INSERT(MAKE-NODE(INITIAL-STATE[*problem*]), *fringe*)

**loop do**

**if** EMPTY?(*fringe*) **then return** failure

*node*  $\leftarrow$  REMOVE-FIRST(*fringe*)

**if** GOAL-TEST[*problem*] applied to STATE[*node*] succeeds  
        **then return** SOLUTION(*node*)

*fringe*  $\leftarrow$  INSERT-ALL(EXPAND(*node*, *problem*), *fringe*)

---

**function** EXPAND(*node*, *problem*) **returns** a set of nodes

*successors*  $\leftarrow$  the empty set

**for each**  $\langle$ action, result $\rangle$  **in** SUCCESSOR-FN[*problem*](STATE[*node*]) **do**

Make-Node {  
        *s*  $\leftarrow$  a new NODE  
        STATE[*s*]  $\leftarrow$  result  
        PARENT-NODE[*s*]  $\leftarrow$  *node*  
        ACTION[*s*]  $\leftarrow$  action  
        PATH-COST[*s*]  $\leftarrow$  PATH-COST[*node*] + STEP-COST(*node*, action, *s*)  
        DEPTH[*s*]  $\leftarrow$  DEPTH[*node*] + 1  
        add *s* to *successors*

**return** *successors*

# Search Strategies

## Uninformed or blind searches:

No information on the length or cost of a path to the solution.

- breadth-first search, uniform cost search, depth-first search,
- depth-limited search, Iterative deepening search, and
- bi-directional search

In contrast: informed or heuristic approaches

# Criteria for Search Strategies

## Completeness:

Is the strategy guaranteed to find a solution when there is one?

## Time Complexity:

How long does it take to find a solution?

## Space Complexity:

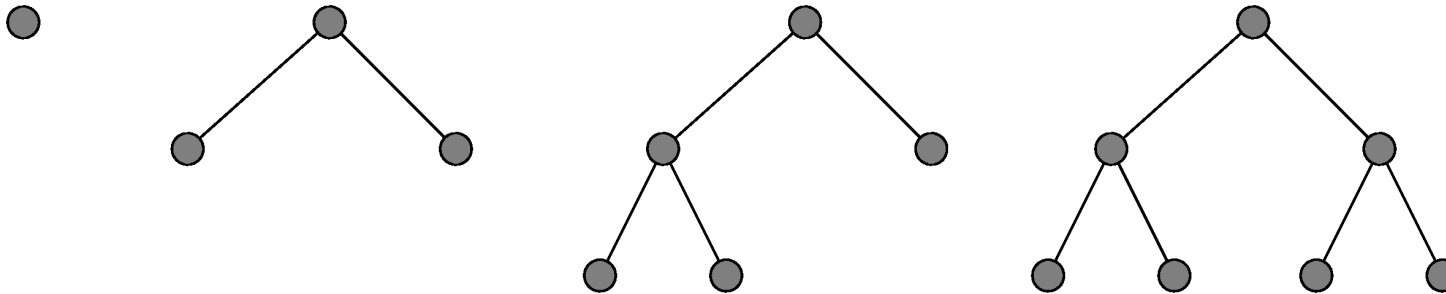
How much memory does the search require?

## Optimality:

Does the strategy find the best solution (with the lowest path cost)?

# Breadth-First Search (1)

Nodes are expanded in the order they were produced . *fringe* = Enqueue-at-end() (FIFO).



- Always finds the **shallowest goal state** first.
- **Completeness**.
- The **solution is optimal**, provided the path cost is a non-decreasing function of the depth of the node (e.g., when every action has identical, non-negative costs).

## Breadth-First Search (2)

The costs, however, are very high. Let  $b$  be the maximal branching factor and  $d$  the depth of a solution path. Then the maximal number of nodes expanded is

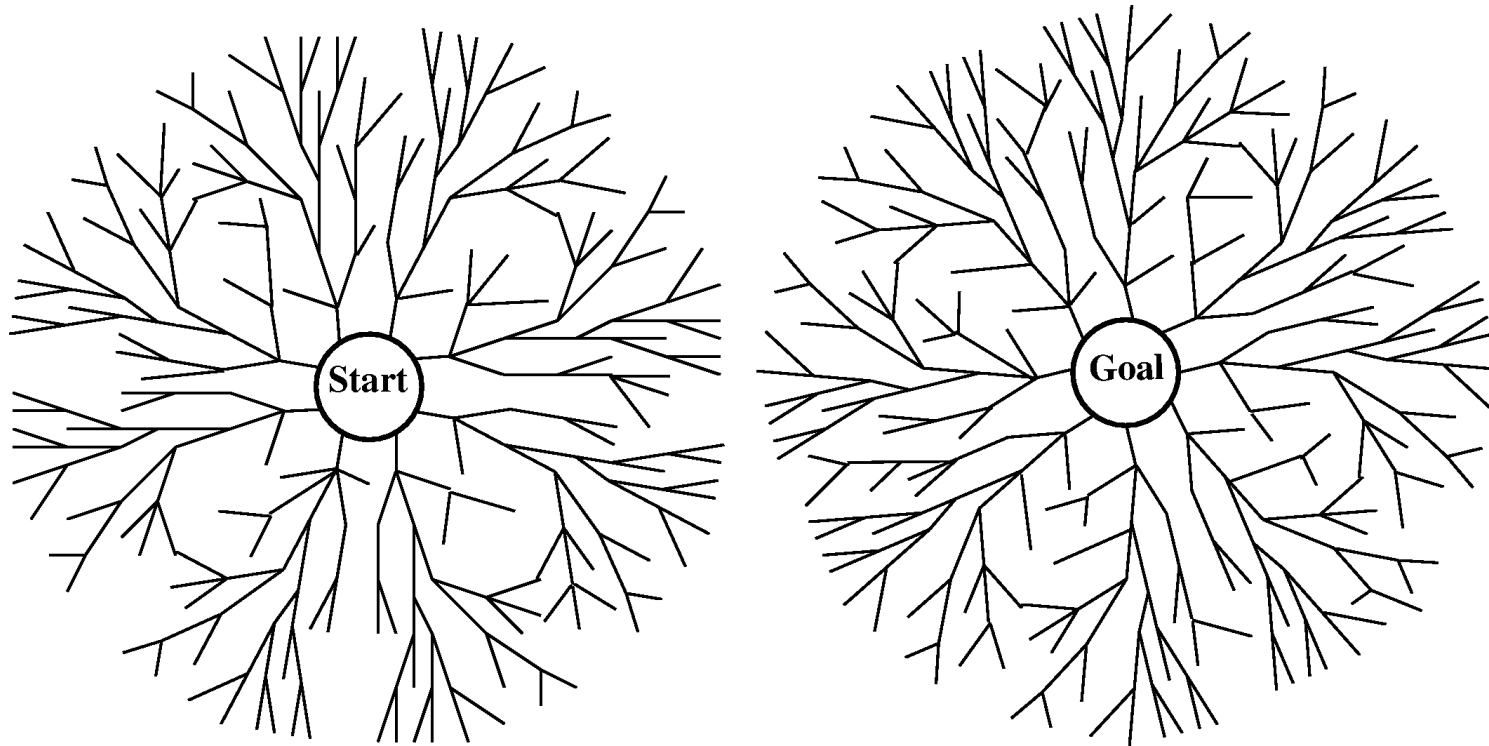
$$b + b^2 + b^3 + \dots + b^d + (b^{d+1} - b) \in O(b^{d+1})$$

Example:  $b = 10$ , 10,000 nodes/second, 1,000 bytes/node:

Depth	Nodes	Time	Memory
2	1,100	.11 seconds	1 megabyte
4	111,100	11 seconds	106 megabytes
6	$10^7$	19 minutes	10 gigabytes
8	$10^9$	31 hours	1 terabyte
10	$10^{11}$	129 days	101 terabytes
12	$10^{13}$	35 years	10 petabytes
14	$10^{15}$	3,523 years	1 exabyte

Note: One could easily perform the goal test BEFORE expansion, then the time & space complexity reduces to  $O(b^d)$

# Bidirectional Search

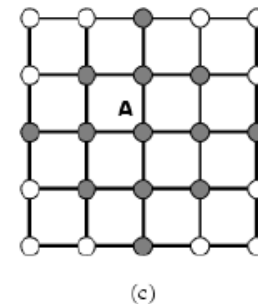
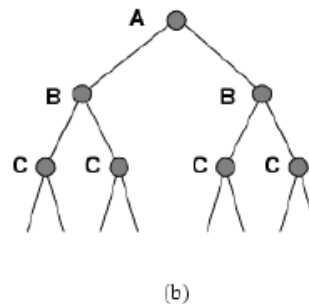
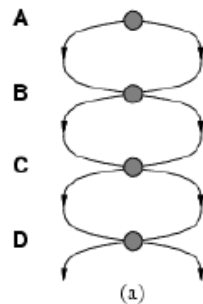


As long as forwards and backwards searches are symmetric, search times of  $O(2 \cdot b^{d/2}) = O(b^{d/2})$  can be obtained.

E.g., for  $b=10$ ,  $d=6$ , instead of 111111 only 2222 nodes!

# Problems With Repeated States

- Tree search ignores what happens if nodes are **repeatedly visited**
  - For example, if actions lead back to already visited states
  - Consider path planning on a grid
- Repeated states may lead to a large (exponential) **overhead**



- (a) State space with  $d+1$  states, where  $d$  is the depth
- (b) The corresponding search tree which has  $2^d$  nodes corresponding to the two possible paths!
- (c) Possible paths leading to A

# Graph Search

- Add a *closed* list to the tree search algorithm
- **Ignore** newly expanded state if already in *closed* list
- *Closed list* can be implemented as **hash table**
- Potential problems
  - Needs a lot of memory
  - Can ignore better solutions if a node is visited first on a suboptimal path (e.g. IDS is not optimal anymore)



# Best-First Search

Search procedures differ in the way they determine the next node to expand.

**Uninformed Search:** Rigid procedure with no knowledge of the cost of a given node to the goal.

**Informed Search:** Knowledge of the cost of a given node to the goal is in the form of an *evaluation function*  $f$  or  $h$ , which assigns a real number to each node.

**Best-First Search:** Search procedure that expands the node with the “best”  $f$ - or  $h$ -value.

# General Algorithm

```
function BEST-FIRST-SEARCH(problem, EVAL-FN) returns a solution sequence
  inputs: problem, a problem
           Eval-Fn, an evaluation function

  Queueing-Fn  $\leftarrow$  a function that orders nodes by EVAL-FN
  return GENERAL-SEARCH(problem, Queueing-Fn)
```

When  $h$  is always correct, we do not need to search!

# Greedy Search

A possible way to judge the “worth” of a node is to estimate its *distance to the goal*.

$$h(n) = \text{estimated distance from } n \text{ to the goal}$$

The only real condition is that  $h(n) = 0$  if  $n$  is a goal.

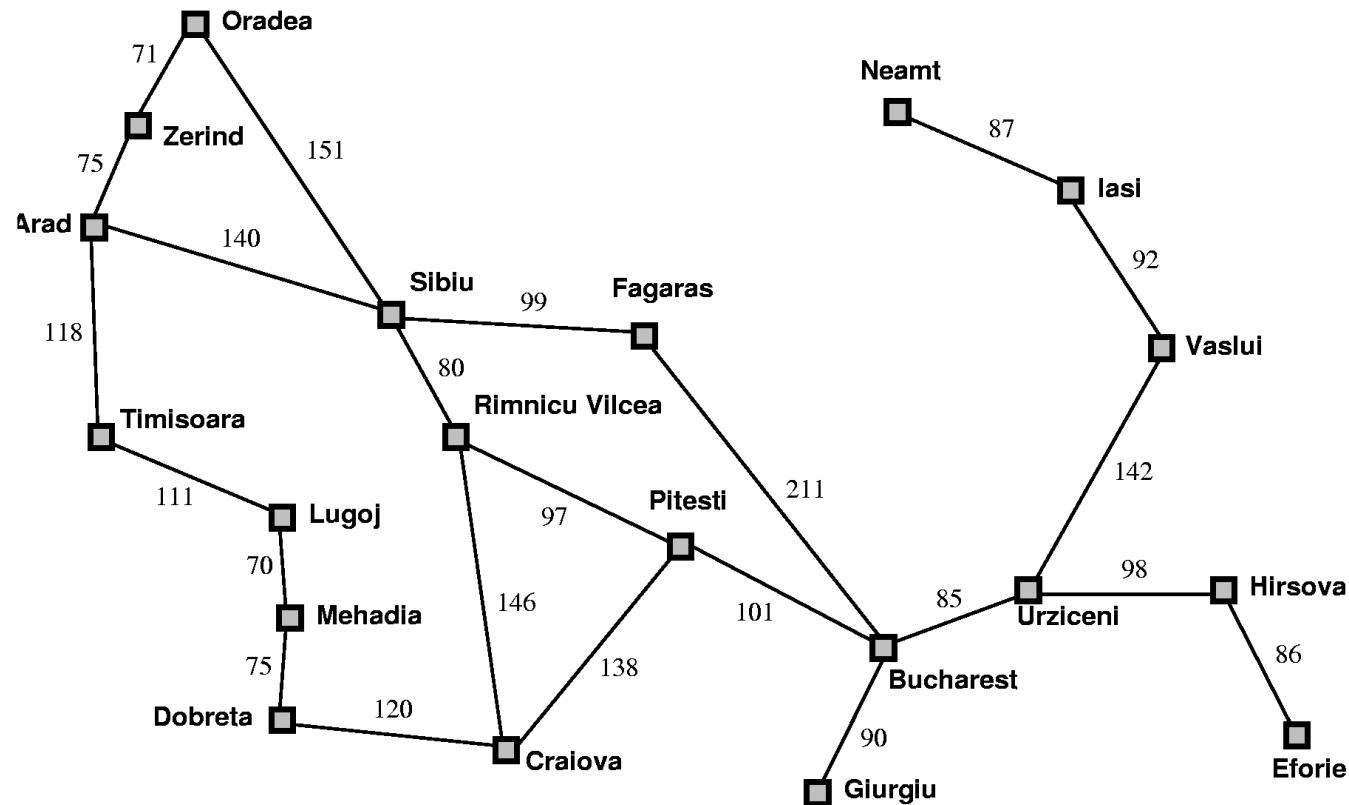
A best-first search with this function is called a *greedy search*.

The evaluation function  $h$  in greedy searches is also called a *heuristic* function or simply a *heuristic*.

→ In all cases, the heuristic is *problem-specific* and *focuses* the search!

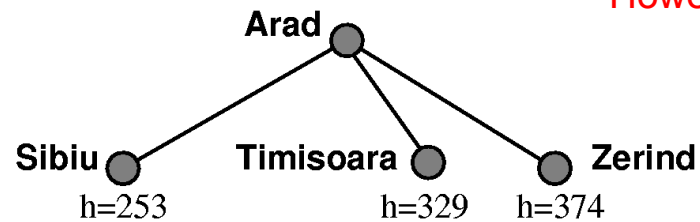
Route-finding problem:  $h$  = straight-line distance between two locations.

# Greedy Search Example

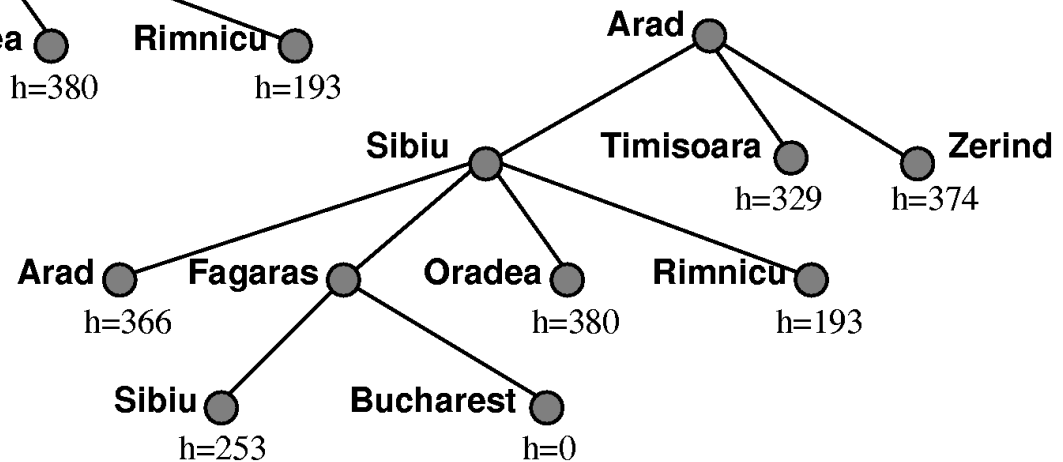
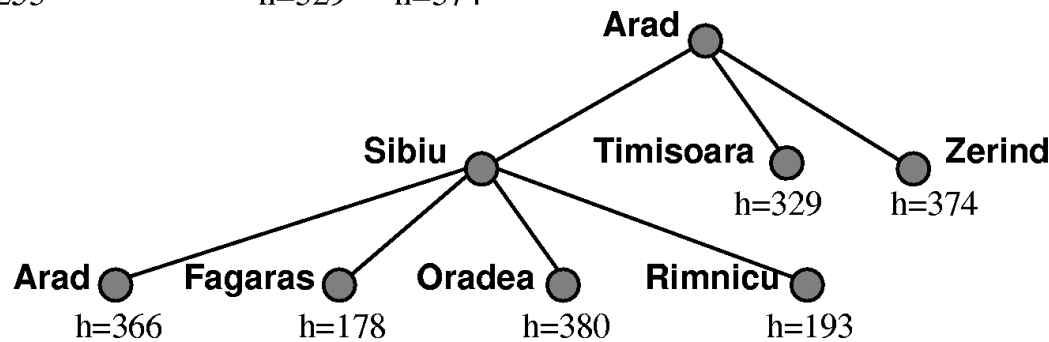


# Greedy Search from *Arad* to *Bucharest*

Arad  
h=366



However: Arad→Sibiu→Fagaras→Bucharest = 450  
Arad→Sibiu→Rimnicu→Pitesti→Bucharest = 418 !



# A\*: Minimization of the estimated path costs

A\* combines the greedy search with the uniform-cost-search, i.e. taking **costs** into account.

$g(n)$  = actual cost from the initial state to  $n$ .

$h(n)$  = estimated cost from  $n$  to the next goal.

$f(n) = g(n) + h(n)$ , the estimated cost of the **cheapest solution** through  $n$ .

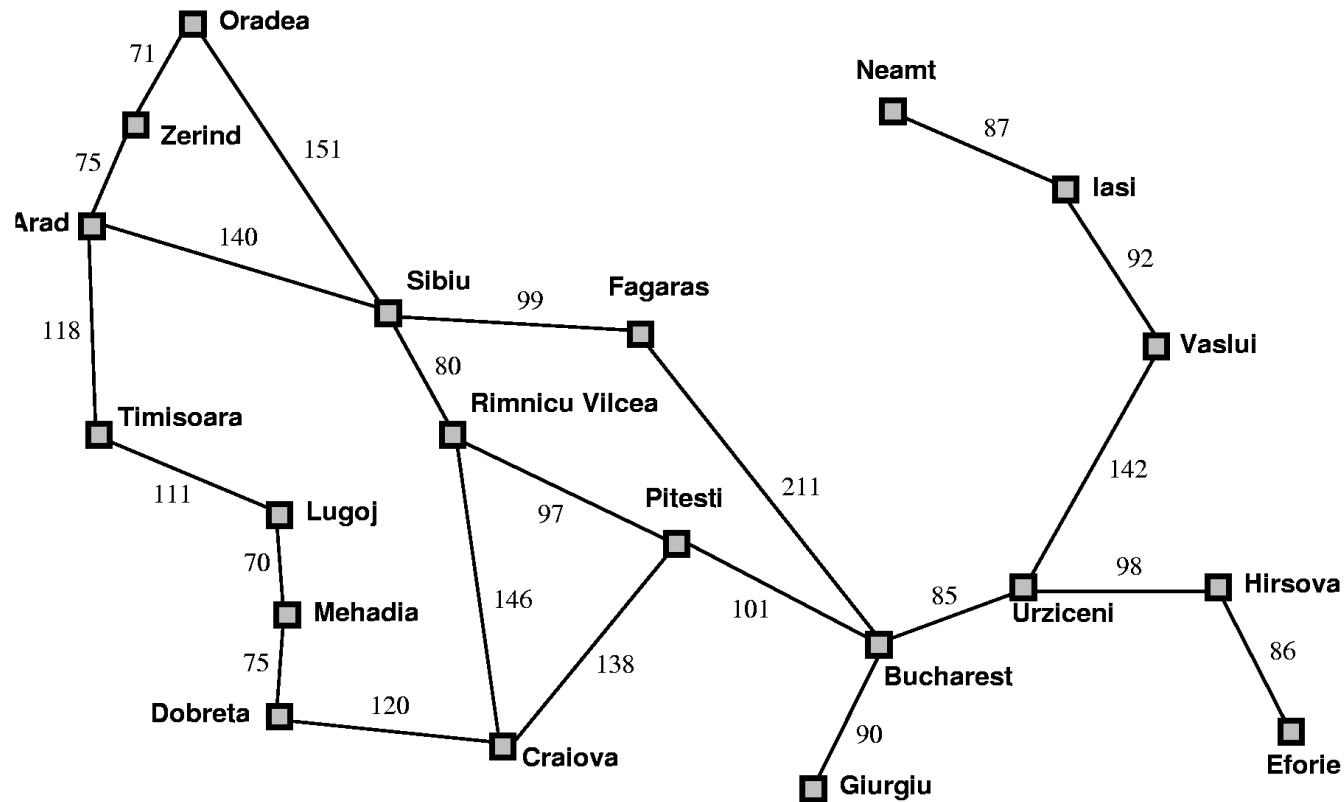
Let  $h^*(n)$  be the **true cost** of the optimal path from  $n$  to the next goal.

$h$  is **admissible** if the following holds for all  $n$  :

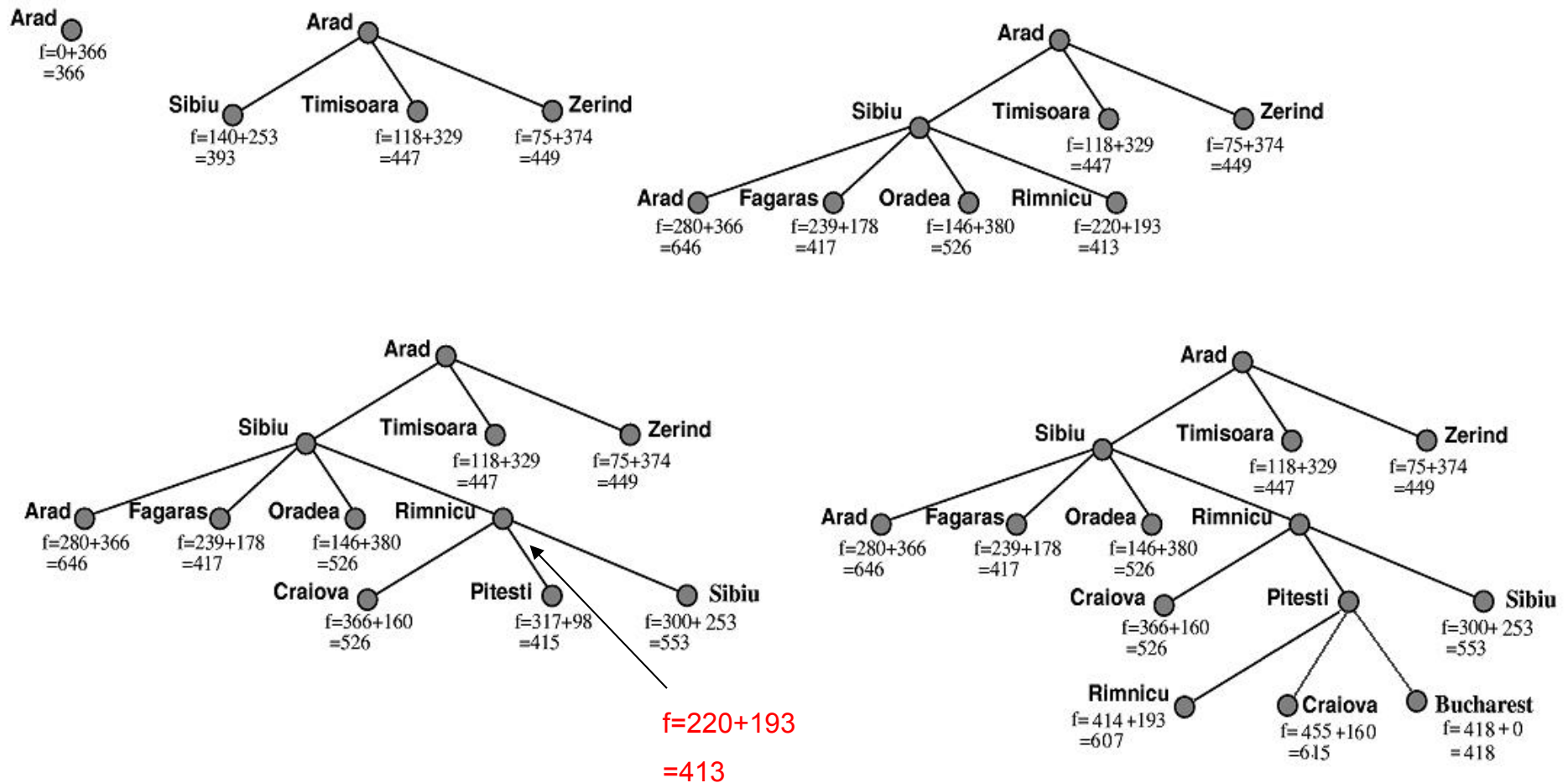
$$h(n) \leq h^*(n)$$

We require that for optimality of A\*,  $h$  is admissible (straight-line distance is admissible).

# A\* Search Example



# A\* Search from *Arad* to *Bucharest*





# Heuristic Function Example

5	4	
6	1	8
7	3	2

Start State

1	2	3
8		4
7	6	5

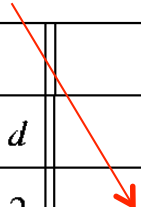
Goal State

- $h_1 =$  the number of tiles in the wrong position
- $h_2 =$  the sum of the distances of the tiles from their goal positions (*Manhattan distance*)

# Empirical Evaluation

- $d$  = distance from goal
- Average over 100 instances
- IDS: Iterative Deepening Search (the best you can do without any heuristic)

# nodes expanded



	Search Cost			Effective Branching Factor		
$d$	IDS	$A^*(h_1)$	$A^*(h_2)$	IDS	$A^*(h_1)$	$A^*(h_2)$
2	10	6	6	2.45	1.79	1.79
4	112	13	12	2.87	1.48	1.45
6	680	20	18	2.73	1.34	1.30
8	6384	39	25	2.80	1.33	1.24
10	47127	93	39	2.79	1.38	1.22
12	364404	227	73	2.78	1.42	1.24
14	3473941	539	113	2.83	1.44	1.23
16	—	1301	211	—	1.45	1.25
18	—	3056	363	—	1.46	1.26
20	—	7276	676	—	1.47	1.27
22	—	18094	1219	—	1.48	1.28
24	—	39135	1641	—	1.48	1.26

# A\* Implementation Details

- How to code A\* efficiently?
- Costly operations are:
  - Insert & lookup an element in the closed list
  - Insert element & get minimal element (f-value) from open list
- The closed list can efficiently be implemented as a hash set
- The open list is typically implemented as a priority queue, e.g. as
  - Fibonacci heap, binomial heap, k-level bucket, etc.
  - binary-heap with  $O(\log n)$  is normally sufficient
- **Hint:** see *priority queue implementation in the "Java Collection Framework"*

## Online search

- Intelligent agents usually don't know the state space (e.g. street map) **exactly** in advance
  - Environment can **dynamically** change!
  - True travel costs are **experienced** during execution
- Planning and plan execution are **interleaved**
- Example: RoboCup Rescue
  - The map is known, but roads might be **blocked** from building collapses
  - Limited drivability of roads depending on **traffic volume**
- Important issue: How to reduce computational cost of **repeated** A\* searches!

# Online search

- Incremental heuristic search
  - Repeated planning of the **complete** path from current state to goal
  - Planning under the **free-space** assumption
  - **Reuse** information from previous planning episodes:
    - Focused Dynamic A\* (**D\***) [Stenz95]
      - Used by DARPA and NASA
    - **D\* Lite** [Koenig et al. 02]
      - Similar as D\* but a bit easier to implement (claim)
  - In particular, these methods reuse **closed list** entries from previous searches
  - All Entries that have been **compromised** by updates (from observation) are adjusted accordingly
- Real-Time Heuristic search
  - Repeated planning with limited **look-ahead** (agent centered search)
  - Solutions can be suboptimal but faster to compute
  - Update of heuristic values of visited states
    - Learning Real-Time A\* (**LRTA\***) [Korf90]
    - Real-Time Adaptive A\* (**RTAA\***) [Koenig06]

# Real-Time Adaptive A\* (RTAA\*)

- Executes A\* plan with limited **look-ahead**
- Learns better informed **heuristic**  $H(s)$  from experience (initially  $h(s)$ , e.g. Euclidian distance)
- Look-ahead defines **trade-off** between optimality and computational cost
- *astar*(lookahead)
  - A\* expansion as far as “lookahead” cells and terminates with state  $s'$

```
while ( $s_{curr} \notin \text{GOAL}$ )  
    astar(lookahead);  
    if ( $s' = \text{FAILURE}$ ) then  
        return FAILURE;  
    for all  $s \in \text{CLOSED}$  do  
         $H(s) := g(s') + h(s') - g(s)$ ;  
    end;  
    execute(plan); //do one step  
    end;  
return SUCCESS;
```

$s'$ : last state expanded during  
previous A\* search

# Real-Time Adaptive A\* (RTAA\*)

## Example

After first A\* planning with  
look-ahead until  $s'$ :

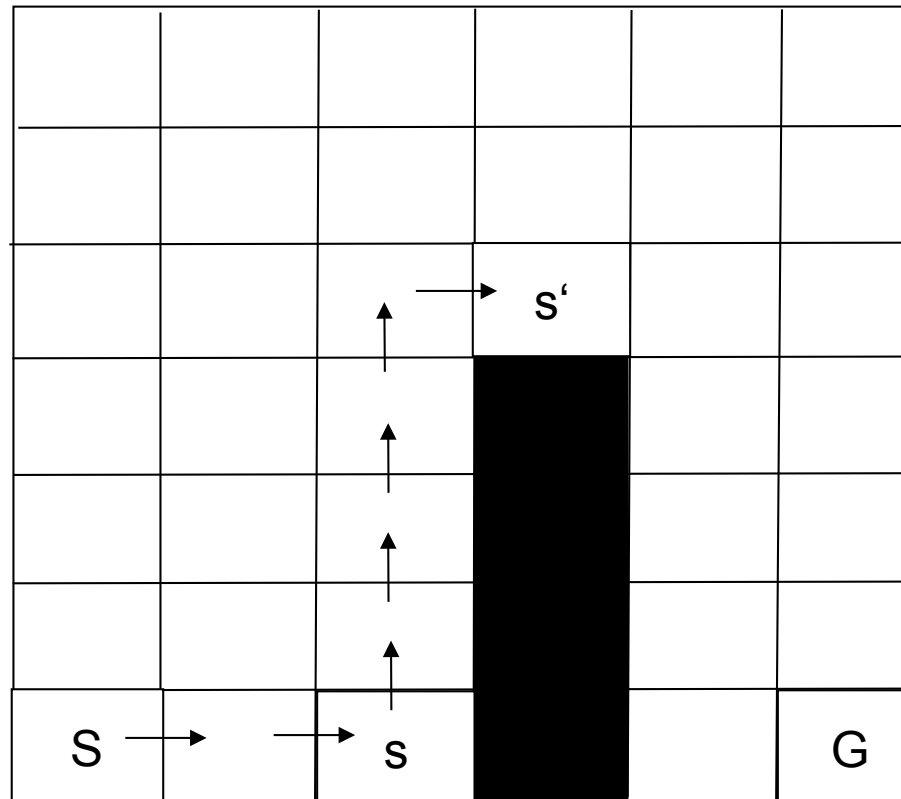
$$g(s')=7, h(s')=6, f(s')=13$$

$$g(s)=2, h(s)=3$$

Update of each element in  
CLOSED list, e.g.:

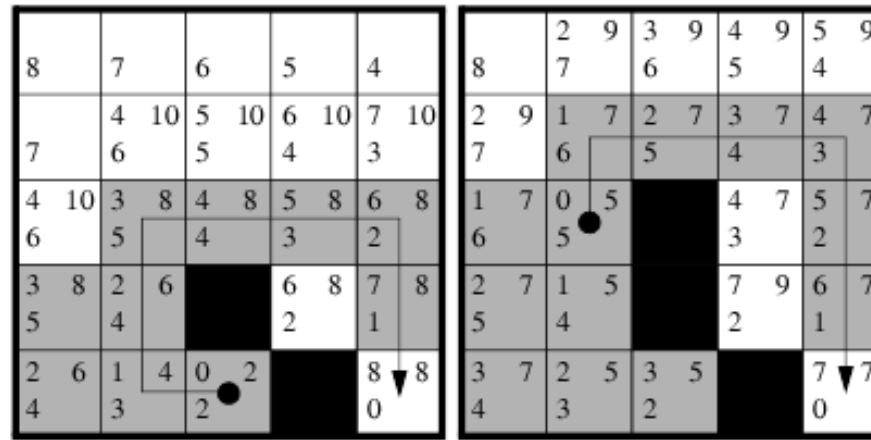
$$H(s) = g(s') + h(s') - g(s)$$

$$H(s) = 7 + 6 - 2 = 11$$

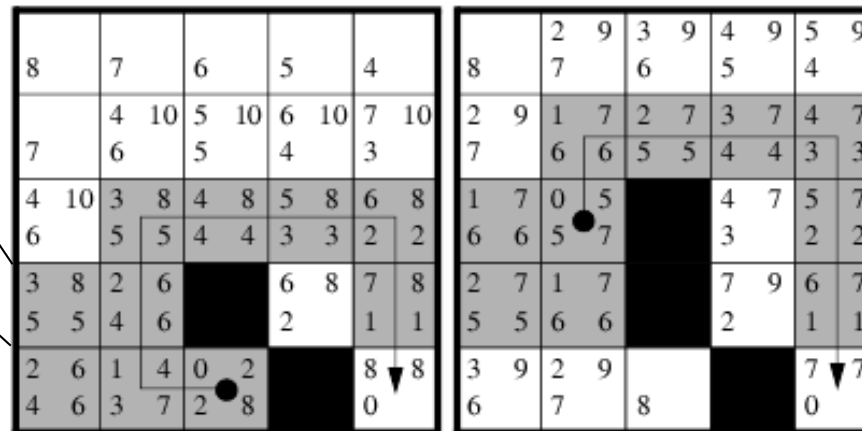
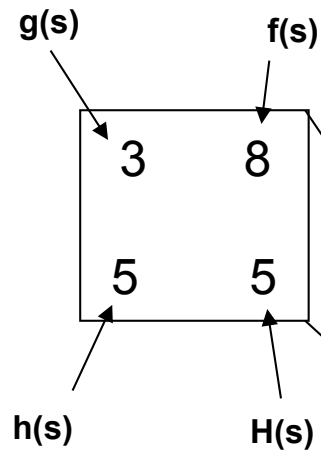


# Real-Time Adaptive A\* (RTAA\*)

A\* vs. RTAA\*



A\* expansion



RTAA\* expansion (inf. Lookahead)



# Case Study: ResQ Freiburg path planner

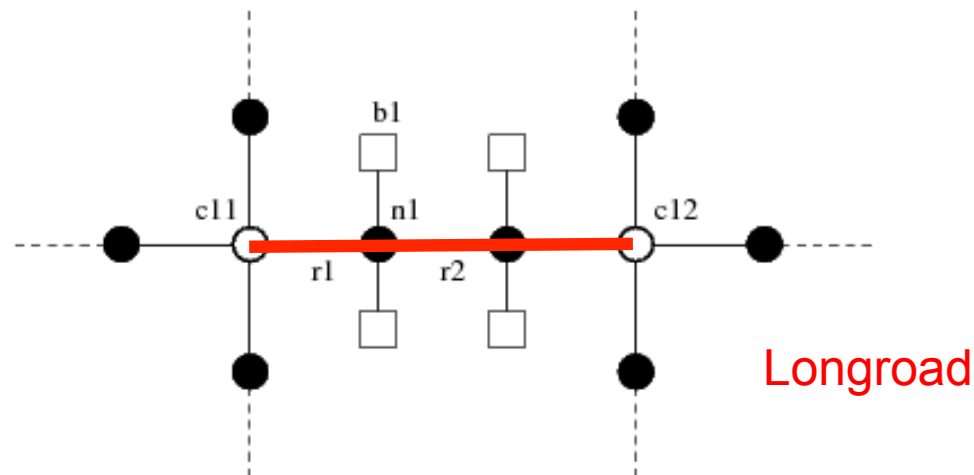
## Requirements

- Rescue domain has some special features:
  - Interleaving between planning and execution is within large time cycles
  - Roads can be merged into “longroads”
- Planner is not used only for path finding, also for supporting task assignment
  - For example, prefer high utility goals with low path costs
  - Hence, planner is frequently called for different goals
- Our decision:
  - Dijkstra graph expansion on longroads
  - Collisions are “reduced” by treating other agents on edges as obstacles (no complete solution)

# Case Study: ResQ Freiburg path planner

## Longroads

- RoboCup Rescue maps **consist** of buildings, nodes, and roads
  - Buildings are directly connected to **nodes**
  - Roads are inter-connected by **crossings**
- For **efficient** path planning, one can extract a graph of **longroads** that basically consists of road segments that are connected by crossings



# Case Study: ResQ Freiburg path planner

## Approach

- Reduction of street network to **longroad** network
- **Caching** of planning queries (useful if same queries are repeated)
- Each agent computes **two** Dijkstra graphs, one for each nearby longroad node
- Selection of optimal path by considering all **4 possible plans**
- Dijkstra graphs are **recomputed** after each perception update (either via direct sensing or communication)
- Additional features:
  - Parameter for favoring **unknown** roads (for exploration)
  - Two more Dijkstra graphs for **sampled time cost** (allows time prediction)

# Case Study: ResQ Freiburg path planner

## Dijkstra's Algorithm (1)

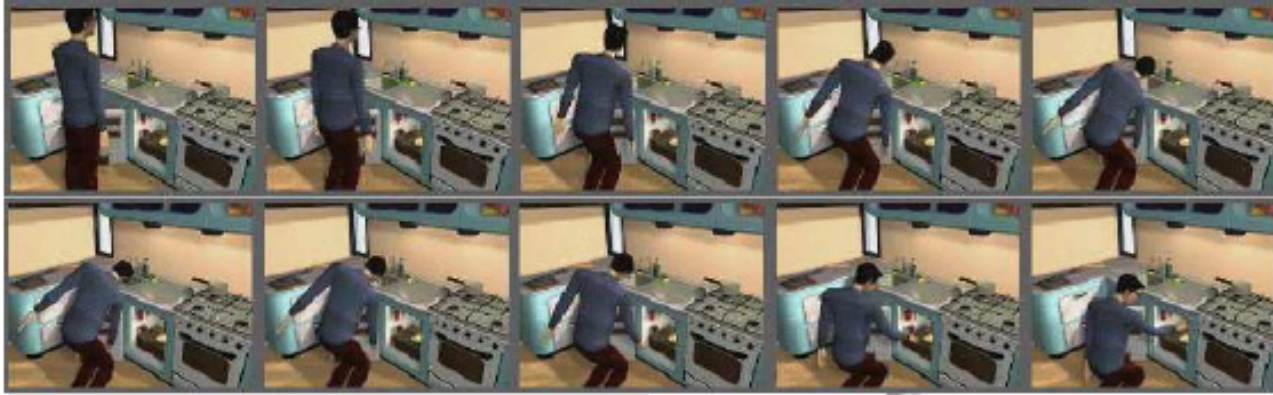
Single Source Shortest Path, i.e. finds the shortest path from a single node to all other nodes

Worst case runtime  $O(|E| \log |V|)$ , assuming  $E > V$ , where  $E$  is the set of edges and  $V$  the set of vertices

- Requires efficient priority queue

# Robot Motion Planning

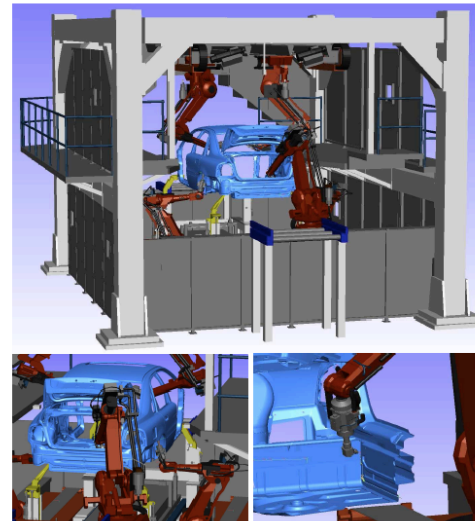
## Introduction



A motion computed by a planning algorithm, for a digital actor to reach into a refrigerator



A planning algorithm computes the motions of 100 digital actors moving across terrain with obstacles



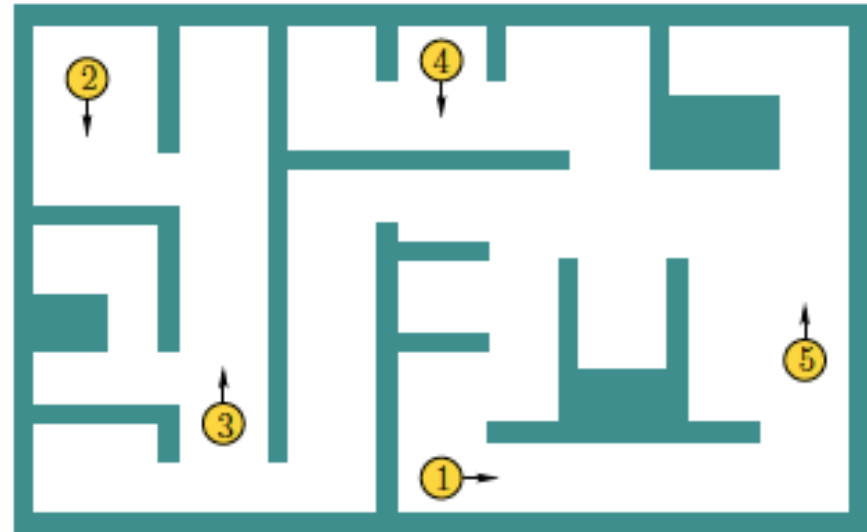
An application of motion planning to the sealing process in automotive manufacturing

# Robot Motion Planning

## Introduction



Using mobile robots to move a piano



Several mobile robots attempt to successfully navigate in an indoor environment while avoiding collisions with the walls and each other

# Robot Motion Planning

## Problem Formulation

The configuration space  $\mathcal{C}$  is the space containing all **possible** configurations of the robot

Suppose world  $\mathcal{W} = \mathbb{R}^2$  or  $\mathcal{W} = \mathbb{R}^3$

Obstacle region  $\mathcal{O} \subset \mathcal{W}$

Rigid robot  $\mathcal{A} \subset \mathcal{W}$

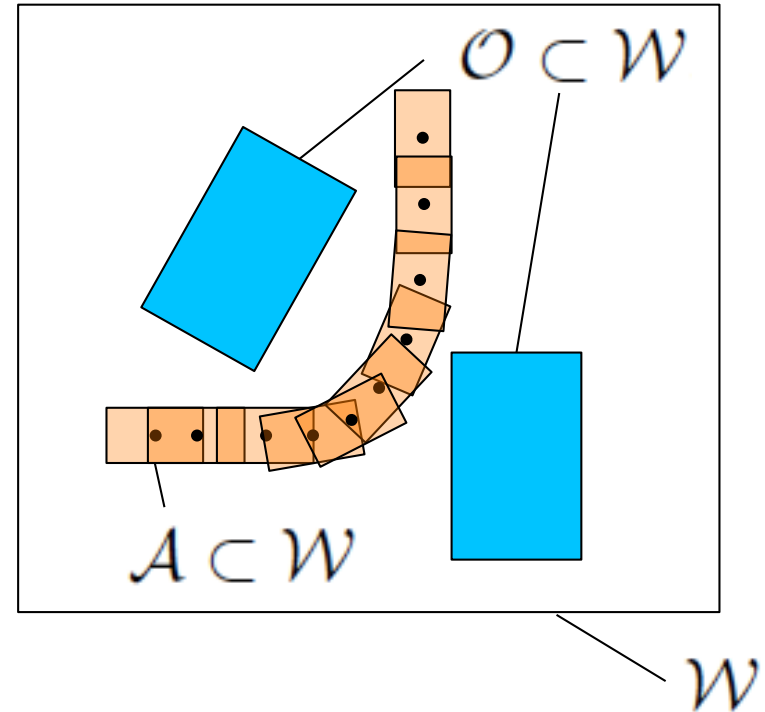
Robot configuration  $q \in \mathcal{C}$

$q = (x_t, y_t, \theta)$  for  $\mathcal{W} = \mathbb{R}^2$

Obstacle region  $\mathcal{C}_{obs} \subseteq \mathcal{C}$  is defined by:

$$\mathcal{C}_{obs} = \{q \in \mathcal{C} \mid \mathcal{A}(q) \cap \mathcal{O} \neq \emptyset\}$$

Which is the set of all configurations  $q$  at which  $\mathcal{A}(q)$ , the transformed robot, intersects  $\mathcal{O}$



The *free space* is defined by:

$$\mathcal{C}_{free} = \mathcal{C} \setminus \mathcal{C}_{obs}$$

# Robot Motion Planning

## Problem / Solution Concepts

**Problem:** Find continuous path  $\tau : [0, 1] \rightarrow \mathcal{C}_{free}$

With  $\tau(0) = c_{start}$  and  $\tau(1) = c_{goal}$

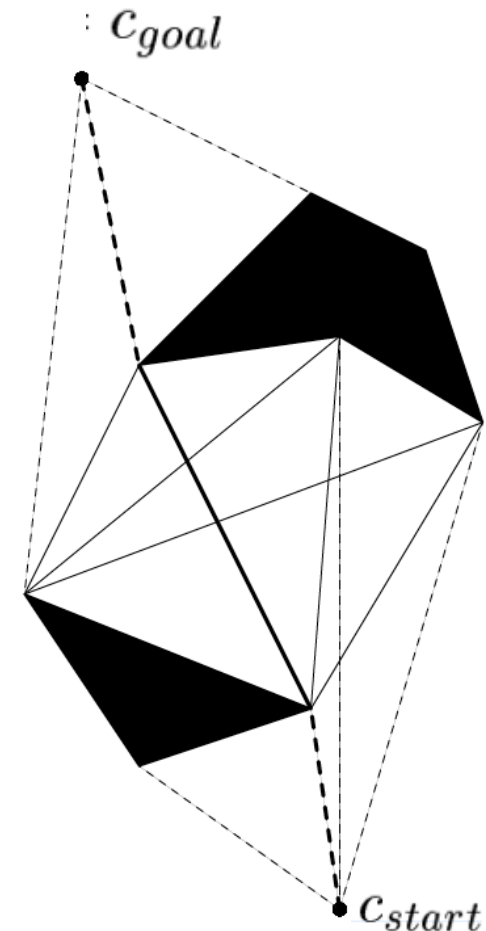
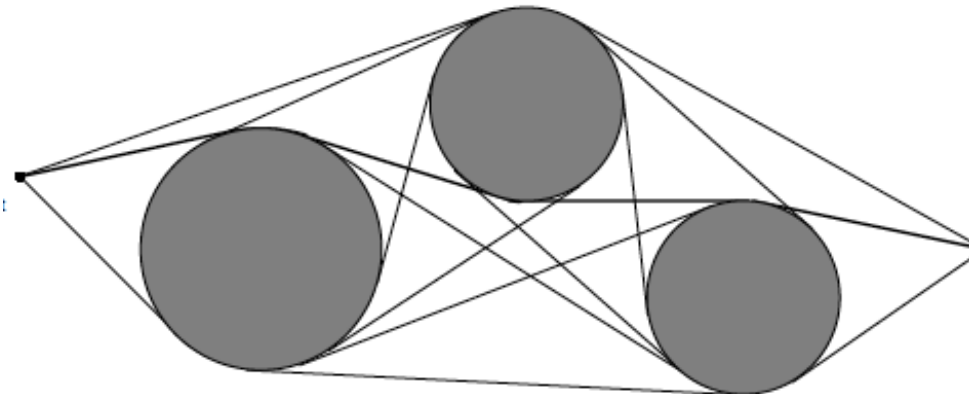
- Requirements
  - Shortest path
  - Minimal **execution time** (requiring a good fit with the motion model, least amount of rotations, etc.)
  - Maximal **distance to obstacles** (needed in dynamic environments, and when sensors are unreliable)
- Many solution concepts, generally we have
  1. Potential Fields (more details in a later lecture)
  2. Visibility Graphs
  3. Grid-based Planning
  4. Sampling-based Planning



# Robot Motion Planning

## Visibility Graphs

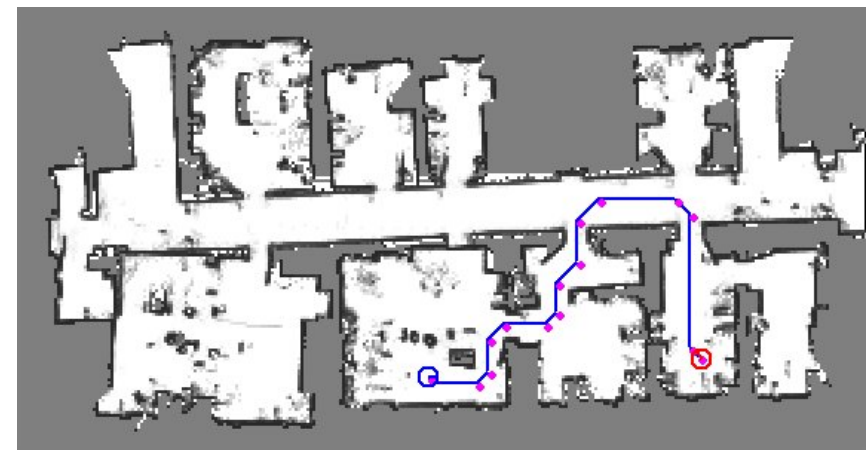
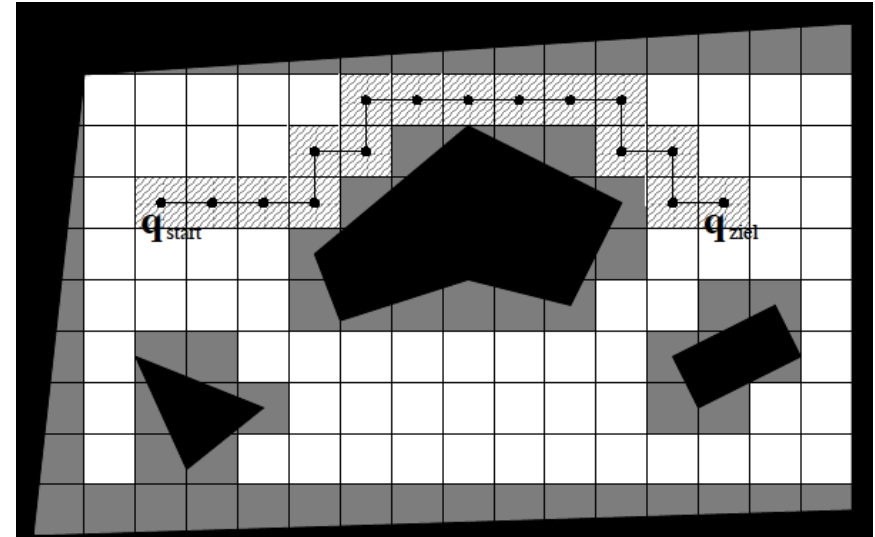
- Approximation of obstacles as **polygons** (or circles)
- Visibility Graph  $S$ : Build visibility graph  $S = (V, G)$ , where  $V$  is the set of all vertices from polygon obstacles or circle tangents
- Planning with **discrete** methods (e.g.  $A^*$ )
- Advantage: Depends **only** on number of **obstacles** only
- Disadvantage: Paths very **close** to obstacles. How to get **good** polygons?



# Robot Motion Planning

## Grid-based Planning

- Planning on a **subdivision** of  $C_{\text{free}}$  into smaller cells
- Simplification: **grow borders** of obstacles up to the diameter of the robot, e.g., by Gaussian blur
- Construction of graph  $G=(V,E)$ , where  $V$  is the set of cells and  $E$  represents their **neighbor-relations**
- Planning with **discrete methods** (e.g.  $A^*$ )
  - Resulting path is a sequence of cells
- **Hierarchical planning**: find path on coarse resolution and re-plan on more fine grained resolutions
- Disadvantage: **Memory** usage grows with the size of the environment
- Advantage: **No polygons!**



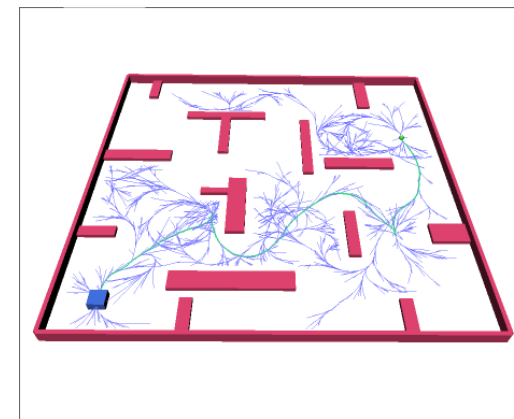
# Robot Motion Planning

## Sampling-based Motion Planning

- **Basic Idea:** To avoid explicit construction of  $C_{obs}$
- Instead: probe  $C_{free}$  with a sampling scheme
- Builds a graph  $G=(V,E)$  by connecting sampled locations
  - each  $e \in E$  has to be collision free!
  - on  $G$  a solution can be found by **discrete** search methods (e.g.  $A^*$ )
- **Critical part:** Random Sampling
- **Time consuming part:** Collision Checks



Sampling without obstacles



Sampling with obstacles

# Sampling-based Motion Planning

## General Procedure

### 1. Initialization:

- Let  $G=(V,E)$  be an undirected search graph with  $(q_{\text{start}}, q_{\text{goal}}) \in V$ ,  $E = \emptyset$

### 2. Vertex Selection Method (VSM):

- Select a vertex  $q_{\text{curr}} \in V$  for **expansion**

### 3. Local Planning Method (LPM):

- Select any  $q_{\text{new}} \in C_{\text{free}}$  by **sampling**
- Find a path  $\tau_s : [0:1] \rightarrow C_{\text{free}}$  such that  $\tau(0) = q_{\text{curr}}$  and  $\tau(1) = q_{\text{new}}$
- $\tau_s$  must be **collision free**, if not, go to 2)

### 4. Insert new Vertex & Edge in the Graph:

- Insert edge between  $q_{\text{curr}}$  and  $q_{\text{new}}$
- Insert  $q_{\text{new}}$  to  $V$

### 5. Check for a Solution:

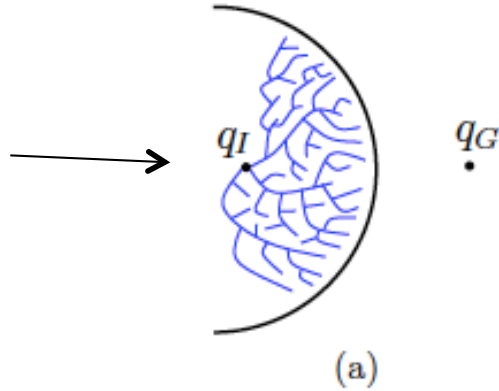
- Check if there **is a valid path** on  $G$  from  $q_{\text{start}}$  to  $q_{\text{goal}}$ , if yes: terminate

### 6. Return to step 2) until any **termination** criterion is met

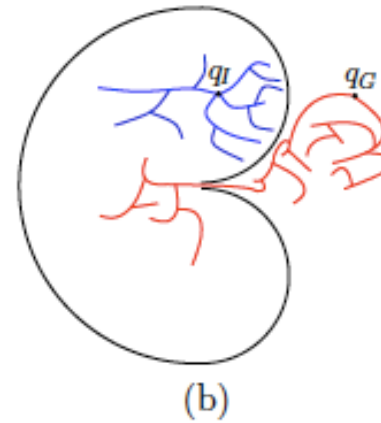
# Sampling-based Motion Planning

## Difficulties

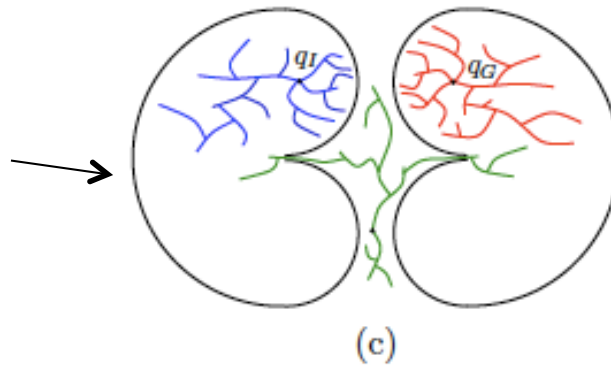
Multi-resolution search required to quickly overcome cavities



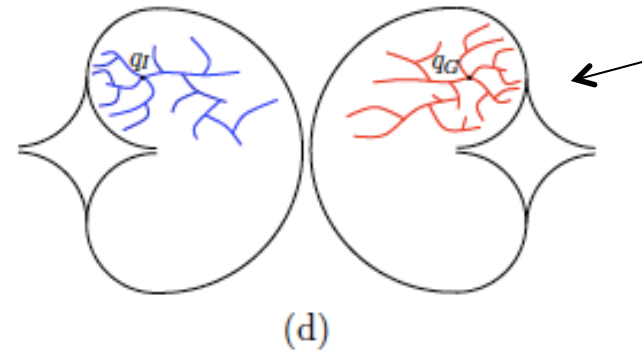
Bidirectional search needed in some cases



Sometimes even multi-dimensional search needed



Hard to solve even with multi-dimensional search



# Sampling-based Motion Planning

## Random Sampling / Deterministic Sampling

- A Sampling sequence should reach **every point** in  $C$ ! However,  $C$  is uncountably infinite ...
- In practice, sampling has to **terminate early**. Hence the sequence of sampling matters!
- **Dense Sequence**: A sequence getting with increasing size arbitrarily close to every element in  $C$
- Random sampling:
  - Suppose  $C=[0,1]$  and  $I \subset C$  is an interval of length  $e$ . If  $k$  samples are chosen independently at random, the probability that none of them falls into  $I$  is  $(1-e)^k$ . As  $k$  approaches infinity, this probability **converges to zero**. This means random sampling is **probably dense**.
- Deterministic sampling:
  - Suppose  $C=[0,1]$  and we want to place 16 samples
  - Simple approach:
    - Select the set  $S=\{i/16 \mid 0 < i < 16\}$  so that all samples **are evenly distributed**
  - What if we want to make  $S$  into a sequence? What is the best ordering? What if 16 points are not enough, i.e., are not reaching every interesting point in  $C$ ?
  - Problem with “sorting by increasing value”: after  $i=8$  half of  $C$  has been neglected! It would be preferable to have a **nice covering of  $C$  for every  $i$**

# Sampling-based Motion Planning

## The Van der Corput sequence

- Idea: to **reverse the order of the bits**, when the sequence is represented with binary decimals
- By reversing the bits, the most significant bit toggles in every step, which means that the sequence **alternates** between the **lower** and **upper** halves of  $C$ .

$i$	Naive Sequence	Binary	Reverse Binary	Van der Corput	Points in $[0, 1]/ \sim$
1	0	.0000	.0000	0	
2	1/16	.0001	.1000	1/2	
3	1/8	.0010	.0100	1/4	
4	3/16	.0011	.1100	3/4	
5	1/4	.0100	.0010	1/8	
6	5/16	.0101	.1010	5/8	
7	3/8	.0110	.0110	3/8	
8	7/16	.0111	.1110	7/8	
9	1/2	.1000	.0001	1/16	
10	9/16	.1001	.1001	9/16	
11	5/8	.1010	.0101	5/16	
12	11/16	.1011	.1101	13/16	
13	3/4	.1100	.0011	3/16	
14	13/16	.1101	.1011	11/16	
15	7/8	.1110	.0111	7/16	
16	15/16	.1111	.1111	15/16	

Sequence for  $i \leq 16$

Note: Both method can also be applied for  $C \subseteq \mathbb{R}^m$  by sampling each dimension **independently**

# Sampling-based Motion Planning

## Rapidly Exploring Dense Trees (RDTs)

Basic algorithm for RDTs  
(without obstacles):

---

**SIMPLE\_RDT**( $q_0$ )

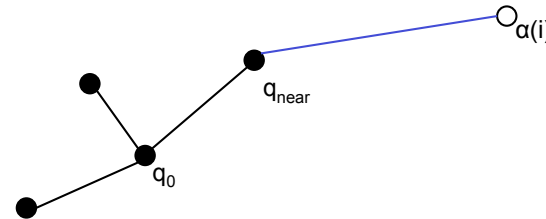
```
1   $\mathcal{G}.\text{init}(q_0);$   
2  for  $i = 1$  to  $k$  do  
3     $\mathcal{G}.\text{add\_vertex}(\alpha(i));$   
4     $q_n \leftarrow \text{NEAREST}(S(\mathcal{G}), \alpha(i));$   
5     $\mathcal{G}.\text{add\_edge}(q_n, \alpha(i));$ 
```

---

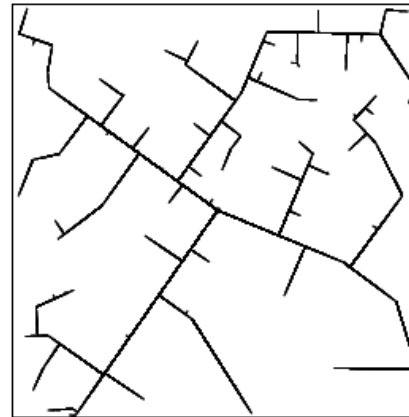
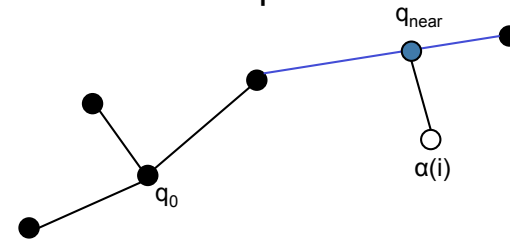
- Let  $S(G)$  be the set of all points reached by  $G$  (either vertices or edges)
- Requires a dense sequence  $\alpha(i)$
- Connects iteratively edges from  $\alpha(i)$  to those nearest in  $G$

Result:

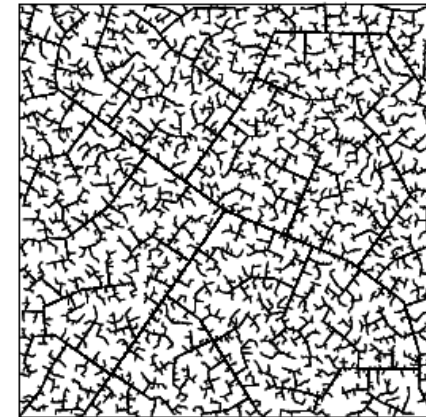
Case 1: Nearest point is a vertex



Case 2: Nearest point is on an edge



45 iterations



2345 iterations



# Sampling-based Motion Planning

## Rapidly Exploring Dense Trees (RDTs)

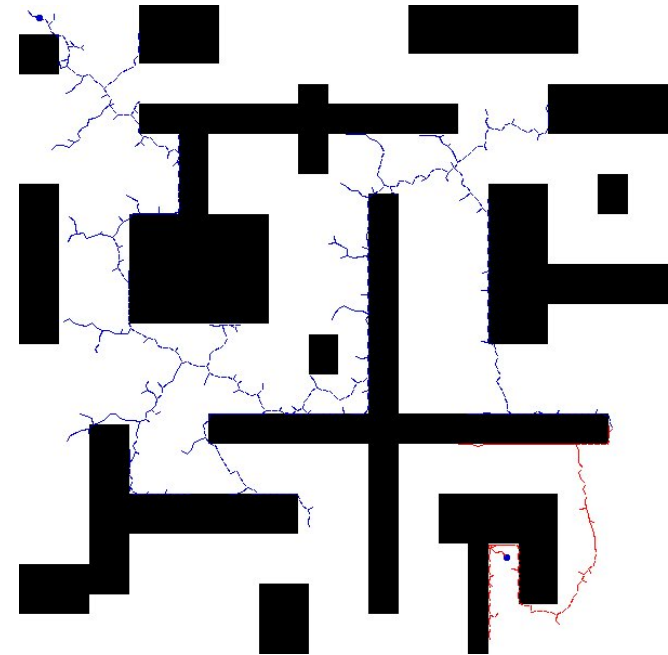
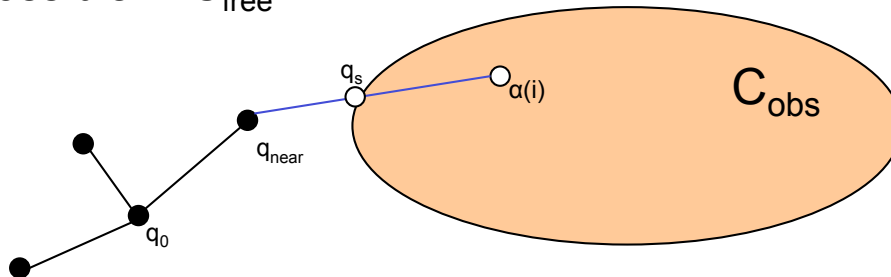
Basic algorithm for RDTs (with obstacles):

---

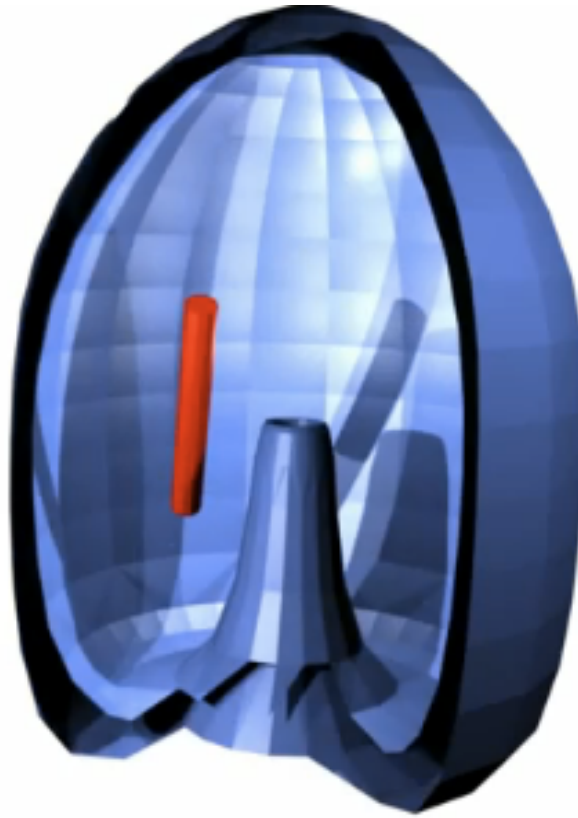
```
RDT( $q_0$ )
1   $\mathcal{G}.\text{init}(q_0)$ ;
2  for  $i = 1$  to  $k$  do
3     $q_n \leftarrow \text{NEAREST}(S, \alpha(i))$ ;
4     $q_s \leftarrow \text{STOPPING-CONFIGURATION}(q_n, \alpha(i))$ ;
5    if  $q_s \neq q_n$  then
6       $\mathcal{G}.\text{add\_vertex}(q_s)$ ;
7       $\mathcal{G}.\text{add\_edge}(q_n, q_s)$ ;
```

---

- STPPING-CONFIGURATION()  
returns the nearest configuration  
possible in  $C_{\text{free}}$



# Bug trap video on YouTube



[http://www.youtube.com/watch?v=qci\\_AktcrD4](http://www.youtube.com/watch?v=qci_AktcrD4)

# Summary

- A problem consists of five parts: The state space, initial situation, actions, goal test, and path costs. A path from an initial state to a goal state is a solution.
- Search algorithms are judged on the basis of completeness, optimality, time complexity, and space complexity.
- Best-first search expands the node with the highest worth (defined by any measure) first.
- When  $h(n)$  is admissible, i.e.,  $h^*$  is never overestimated, we obtain the A\* search, which is complete and optimal.
- Online search provides method that are computationally more efficient when planning and plan execution are tightly coupled
- Sampling-based Planning methods are well suited for Robot Motion Planning

# Literature

- Homepage of Tony Stentz:
  - A. Stentz **The focussed D\* algorithm for real-time** replanning Proc. of the Int. Join Conference on Artificial Intelligence, p. 1652-1659, 1995.
- Homepage of Sven Koenig:
  - S. Koenig and X. Sun. **Comparing Real-Time and Incremental Heuristic Search for Real-Time Situated Agents** Journal of Autonomous Agents and Multi-Agent Systems, 2009
  - S. Koenig and M. **Likhachev Real-Time Adaptive A\*** Proceedings of the International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS), 281-288, 2006
  - S. Koenig and M. Likhachev. **Fast Replanning for Navigation in Unknown Terrain** Transactions on Robotics, 21, (3), 354-363, 2005.
- More difficult to find, also explained in the AIMA book (2nd ed.):
  - R. Korf. **Real-time heuristic search**. Artificial Intelligence, 42(2-3):189-211, 1990.
  - Demo search code in Java on the **AIMA** webpage <http://aima.cs.berkeley.edu/>