

EGN 4060c: Introduction to Robotics

Lecture 8: Path Planning I

Instructor: Dr. Gita Sukthankar

Email: gitaras@eecs.ucf.edu

Announcements

- Lab report for Lab 2 due next Wed.
- Extensions of one week are available to people who didn't manage to finish the lab this week.
- No class on Sept 17th but Astrid will open the lab for teams that are still working on completing Lab 2.
- Lab 3: Path Planning begins on Tuesday
 - Write a path planner to navigate robot through a maze using an existing map
 - Move the robot to follow the existing path
- Reading on planning in Chapter 9 and 10

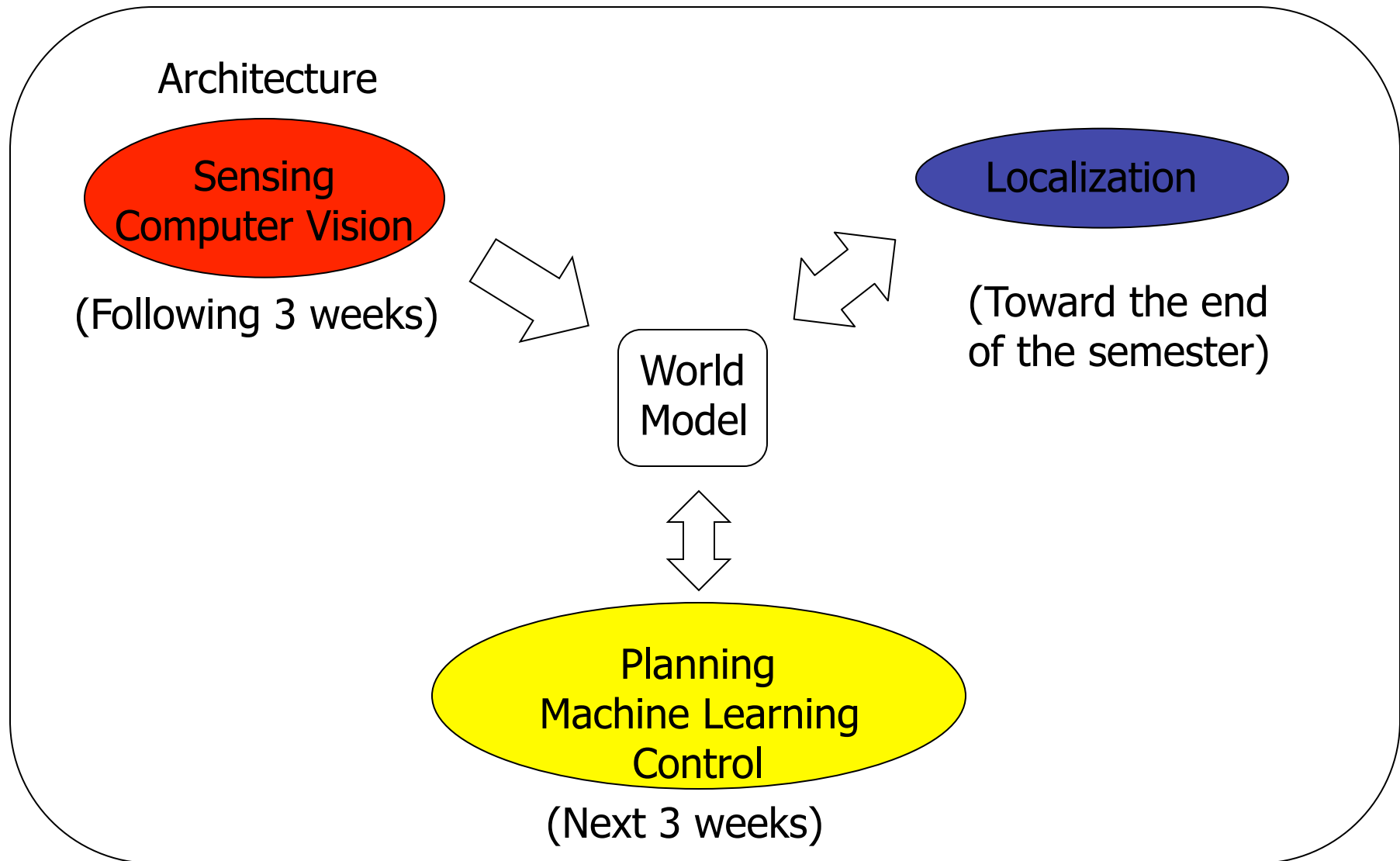
Lab Reports

- Only one lab writeup should be submitted per team
- The lab writeup should start with one or two paragraphs summarizing the general operation of the code
- Include a javadoc style description of the new classes and methods you created for the lab.
- If most of your code is within one method, include a summarized version of the critical section of the code in your report.
- At the conclusion, you should add a couple of sentences mentioning any significant problems you experienced during the lab. You won't be graded on this section; it's just a way for me to understand which aspects of the assignment are giving the students trouble.
- On average, writeups should be between 2-5 pages long. The labs at the beginning of the semester will be simpler and will require a shorter writeup.
- Using figures or diagrams to explain the operation of your system are always welcome.
- The writeups will be graded on a scale of 1-3 pts; points will be awarded for clarity and detail.
- In addition to your writeup, please submit any new java files that you wrote for the lab.

Lab Report Tips

- Being organized about your thought process and your writing is the best way to do well on lab reports.
- Don't forget to include anything that was asked for in the lab instructions (e.g. in this lab tables of measurements)
- Do not write a chronological narrative of what you did in lab.
- Proof-read your document
- Imagine that you were the reader of the document---would you be able to understand what was done?

Where Planning Fits In.....



How to select a planner?

- There are lots of different types of planning methodologies.
- How do you decide which method to use?

Can the robot

- Identify objects and their absolute location with confidence?
- Determine its absolute location in the world?
- Execute actions reliably?

How uncertain is the world?

Does the environment....

- Require a large number of states to describe it (lots of objects, locations, actions). This is known as having a large state space.
- Are the control actions described with a high dimensional vector?
- Change rapidly due to events not caused by the robot (exogeneous events). This is often referred to as being a dynamic environment.

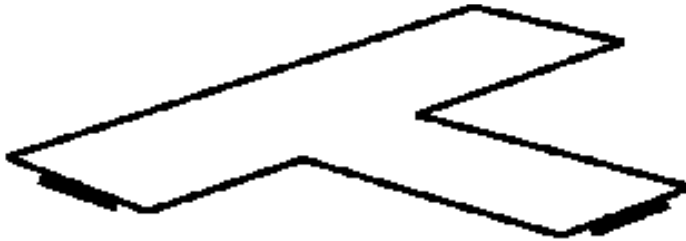
Localization

- Why is it difficult for the robot to figure out where it is in the world?

Localization

- Limitations of encoders
 - Slippage!
- Limitations of sensing technology

Types of World Models



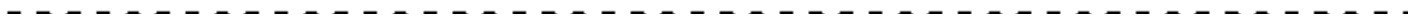
Metric: distances, directions, shapes in coordinate system



Topological: connectivity



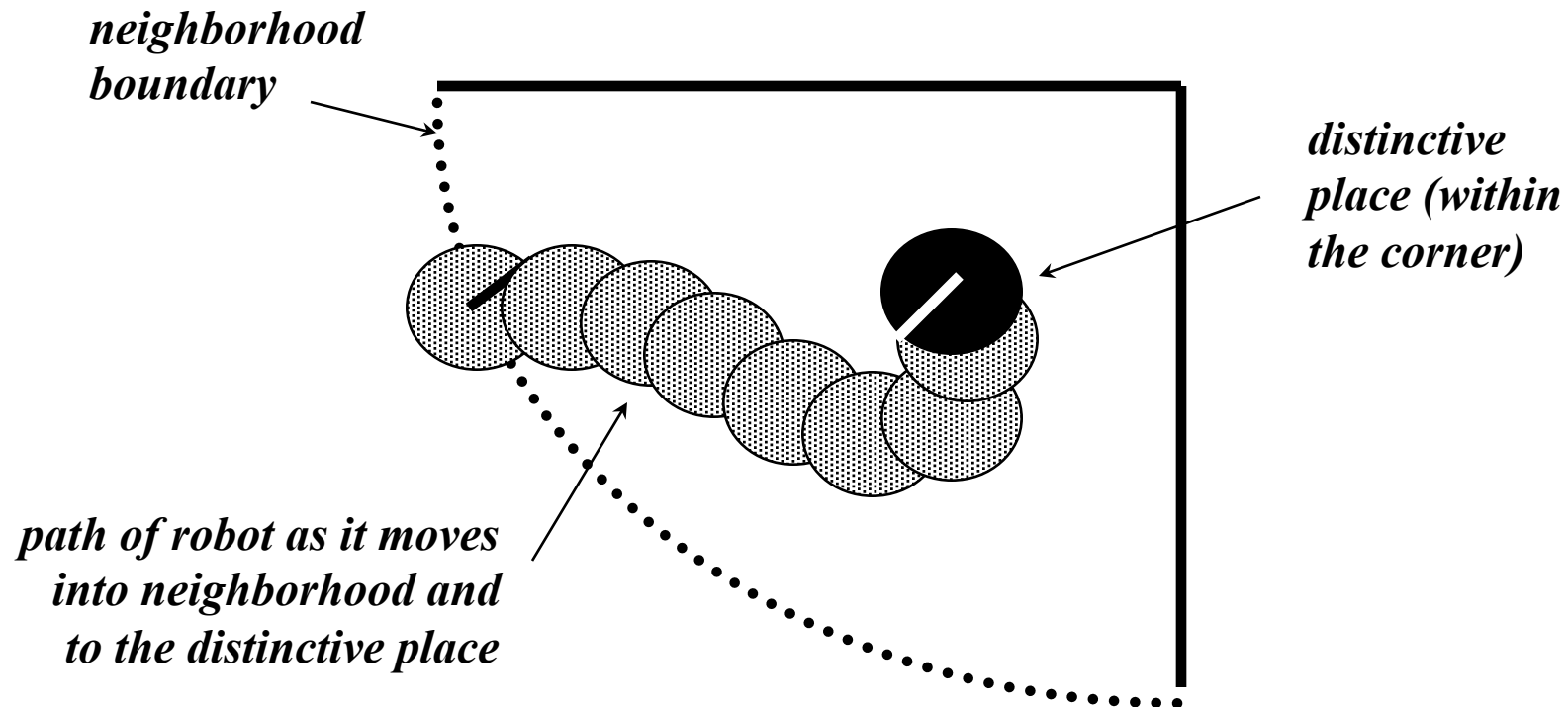
Landmark definitions, procedural knowledge for traveling



What does the metric map allow you to do that the topological map does not?

Hint: it's all about distance!

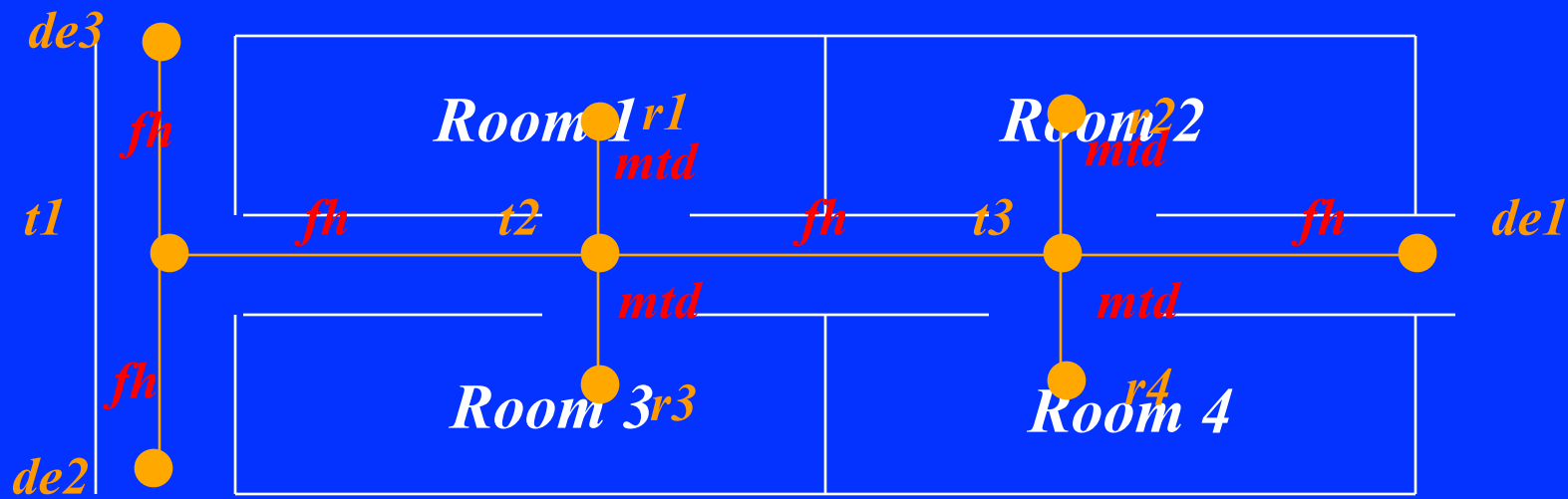
Old-Style: Using Landmarks



Use one behavior until robot sees the distinctive place (exteroceptive cueing) then swap to a landmark localization behavior

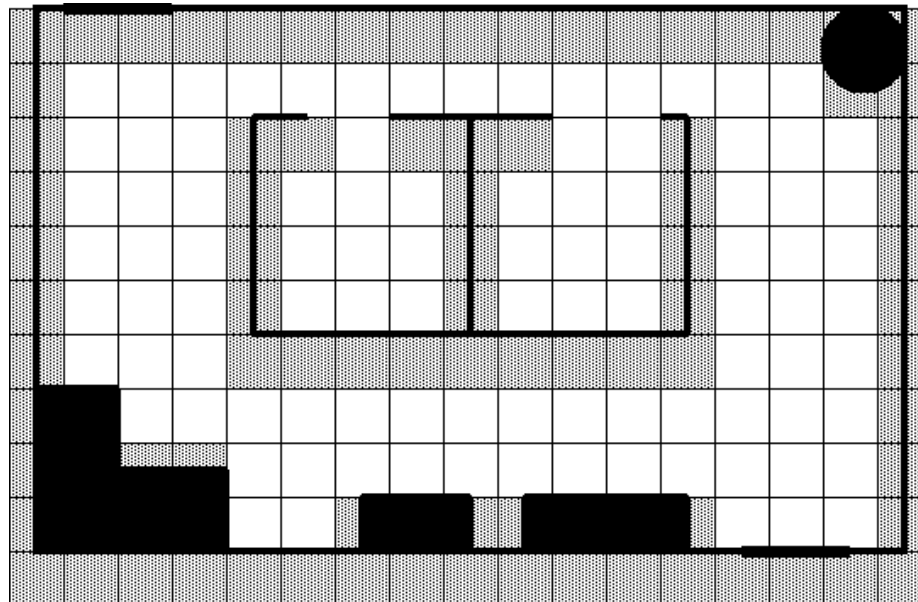
Relational Graphs

- Convert floorplan into the relational graph before giving it to the planner
- Label each edge with the appropriate local control schema: move through door, follow hall
- Label each node with the type of gateway: dead-end, junction, room
- Use a graph search algorithm like Dijkstra's



Occupancy Grids

- Most commonly used metric map representation
 - Discretize the world
 - Make a relational graph by each element as a node, connecting neighbors (4-connected, 8-connected)
 - Requires more memory than the other representation



MapGUI: class for map display

```
MapGUI map = new MapGUI();  
map.getMap();  
map.moveRobot(row, column, angle);  
map.moveRobot(row, column, direction);  
map.getRobotLocation();  
Example map file format: 0=empty, 1=wall, 2=goal
```

```
6 8  
00100010  
00000000  
00000100  
01101100  
00011000  
10010002  
2 1 120
```

Overview

- Search-based planning: A*
- Wavefront planner
- Replanning
- Mission planning: non-navigational planners

Next time:

- Configuration space
- Visibility and Voronoi diagrams
- High-dimensional planners
- Sensor-based planners

Planning as Search

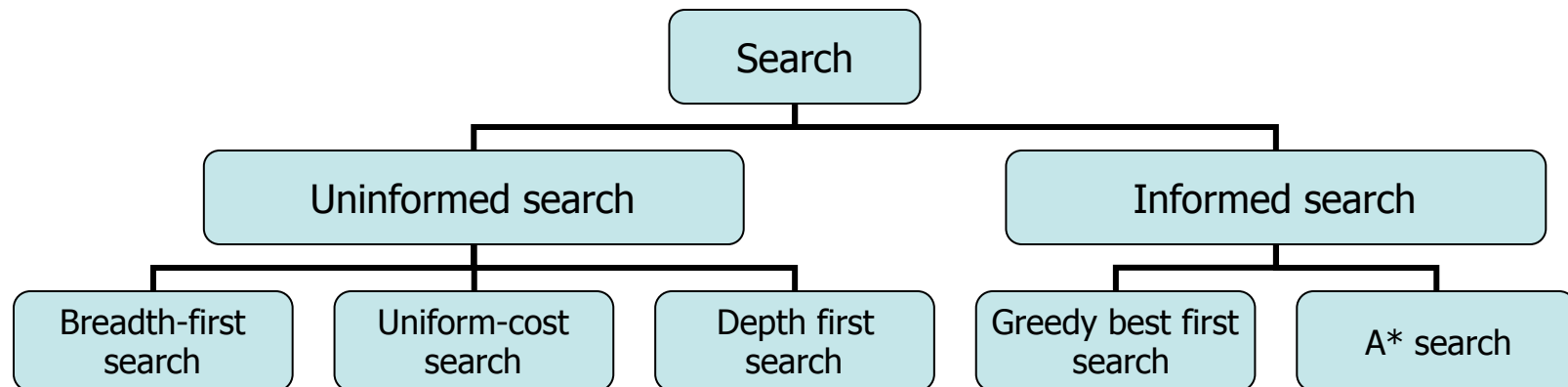
- Find a sequence/set of actions that can transform an ***initial state*** of the world to a ***goal state***
- Planning Involves ***Search*** Through a ***Search Space***
 - How to represent the search space? (occupancy grid)
 - How to conduct the search? (search heuristics)
 - How to evaluate the solutions? (cost functions)

Search State Space

- Each node has a set of successor nodes
- The process of expanding a node means to generate all of the successor nodes and add them and their associated arcs to the state-space graph
- One or more nodes are designated as start nodes
- A goal test determines if a node is a goal node
- A solution is a path in a state space from a start node to a goal node
- The cost of a solution is the sum of the arc costs on the solution path

Search Recap

- A sampling of search strategies ... there are many more



To use informed search, you need to have some way of evaluating the nearness of a state to the goal state (possible with metric representations)

Planning as Search

- Uninformed search:
 - Breadth-first search: queue (FIFO)
 - Depth-first search: stack (LIFO)
- Informed search:
 - A*: priority queue based on costs to get to a location plus estimate of how much further to go

A* requires an admissible heuristic to be efficient.

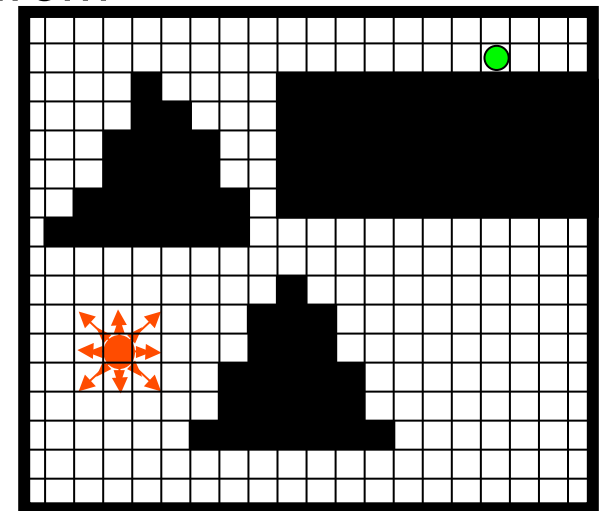
Once we have our state space (and action space, and cost function...)

- Perform tree-based search

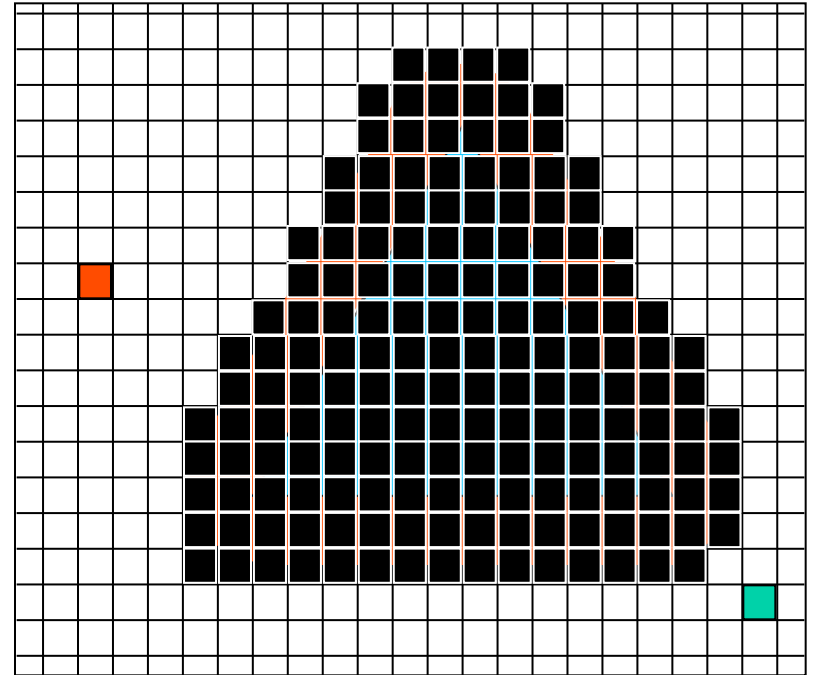
- Construct the root of the tree as the start state, and give it value 0
- While there are unexpanded leaves in the tree
 - Find the leaf x with the lowest value
 - For each action, create a new child leaf of x
 - Set the value of each child as:

$$g(x) = g(\text{parent}(x)) + c(\text{parent}(x), x)$$

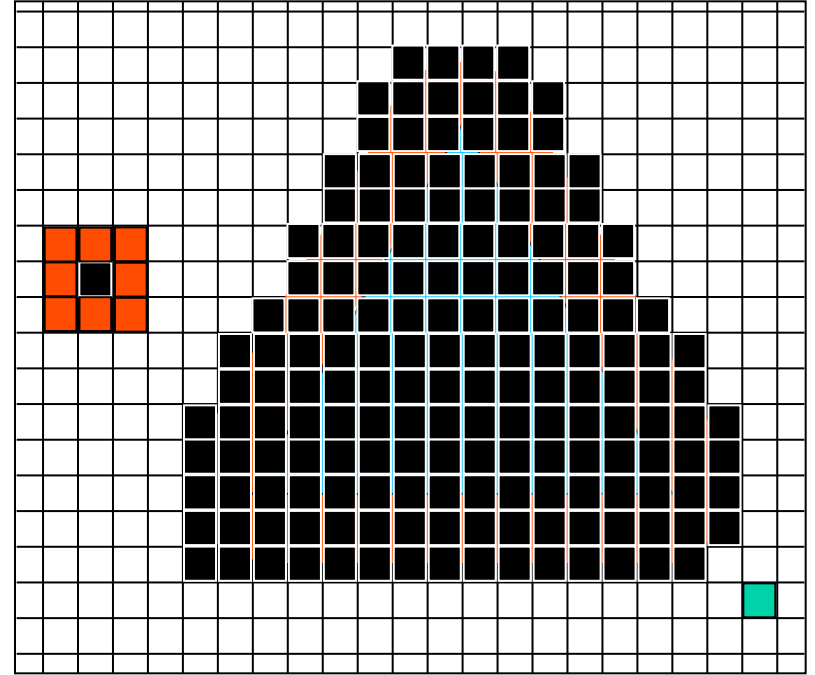
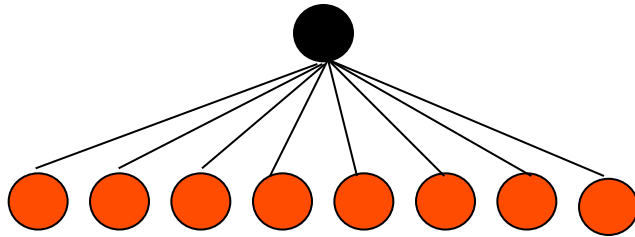
where $c(x, y)$ is the cost of moving from x to y (distance, in this case)



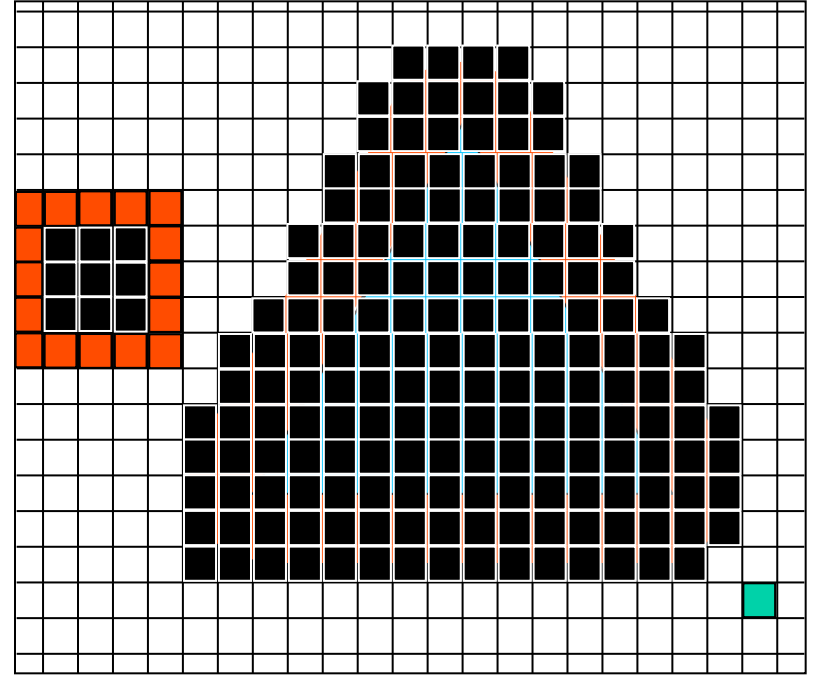
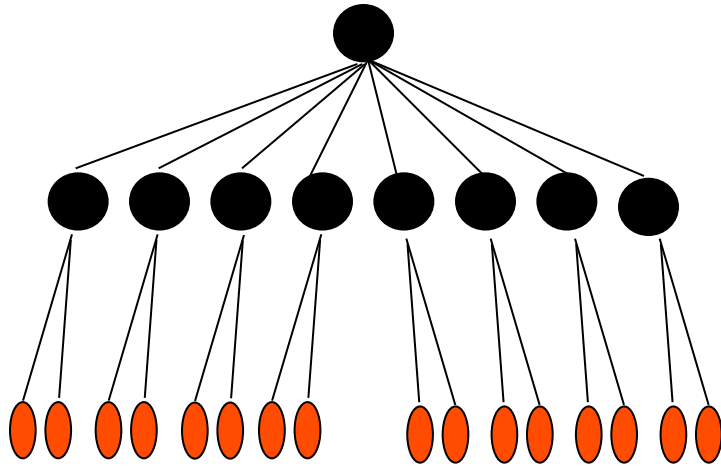
Planning by Searching a Tree



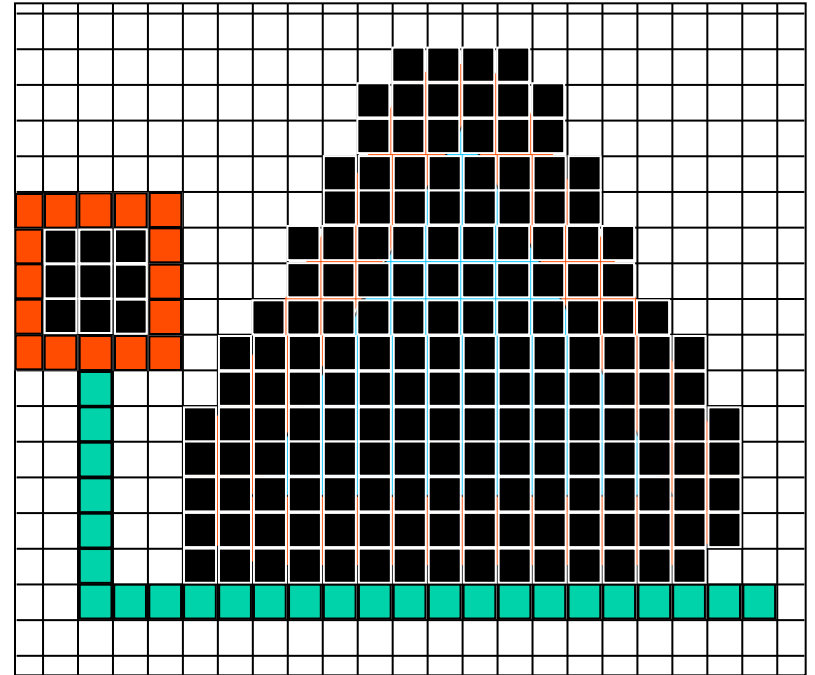
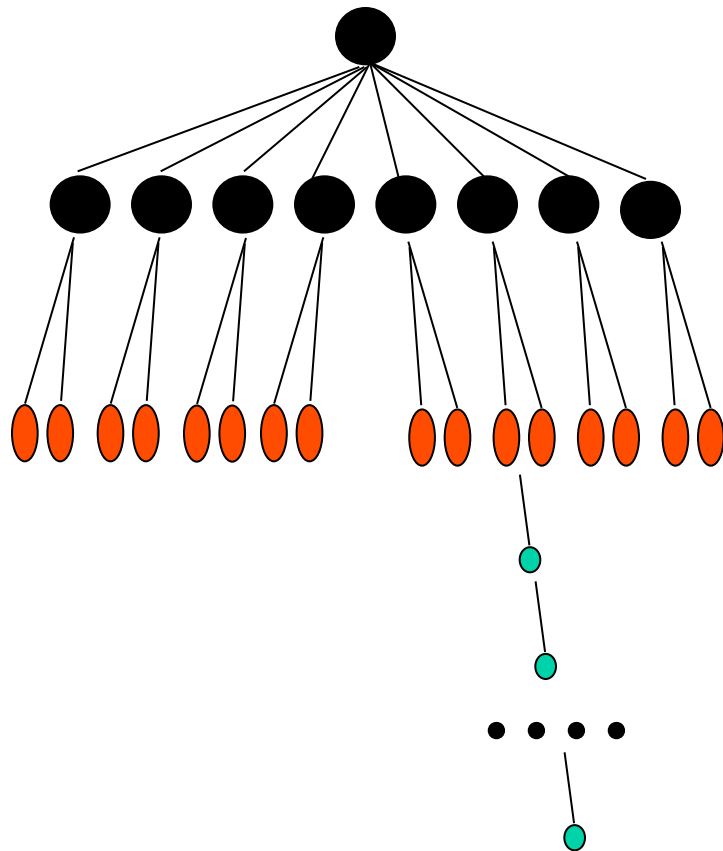
Planning by Searching a Tree



Planning by Searching a Tree



Planning by Searching a Tree



Informed Search – A*

- Use domain knowledge to bias the search
- Favour a path that is closer to the

Cost incurred
from the start state

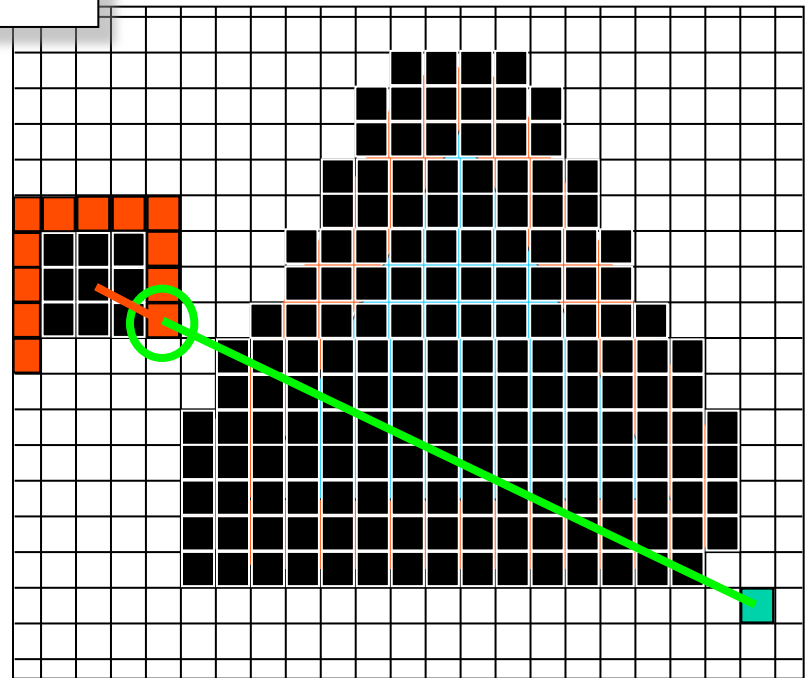
Estimated cost from
here to the goal:
“heuristic” cost

- Each state gets a value

$$f(x) = g(x) + h(x)$$

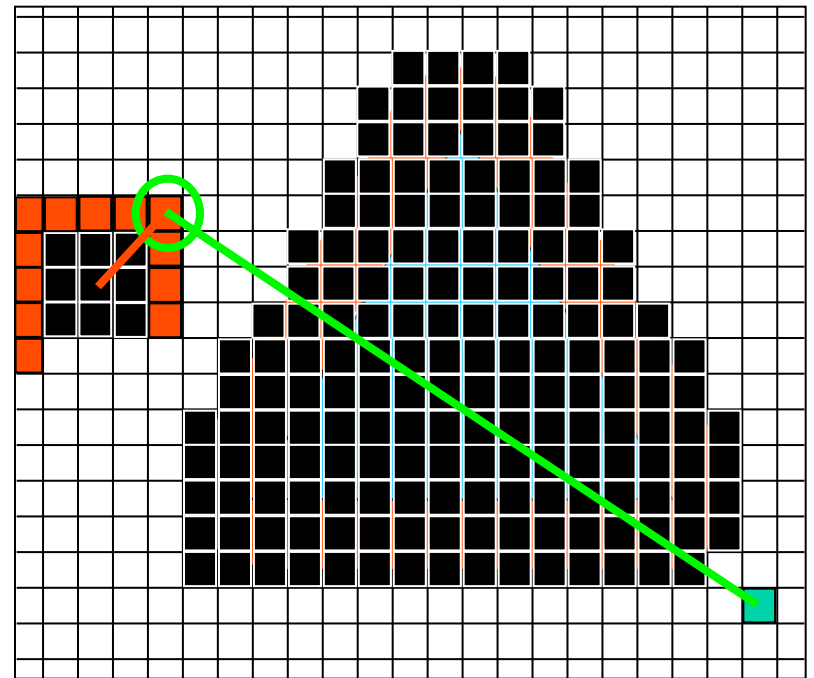
- For example

- $g(x) = 2$
- $h(x) = \text{Euclidean distance}$
 $= \sqrt{8^2 + 17^2}$
 $= 18.8$
- $f(x) = 20.8$



Informed Search – A*

- Use domain knowledge to bias the search
- Favour actions that might get closer to the goal
- Each state gets a value $f(x)=g(x)+h(x)$
- For example
 - $g(x) = 2$
 - $h(x) = \text{Euclidean distance}$
 $= \text{sqrt}(11^2 + 17^2)$
 $= 20.2$
 - $f(x) = 22.2$



How to choose heuristics

- The closer $h(x)$ is to the true cost to the goal, $h^*(x)$, the more efficient your search

BUT

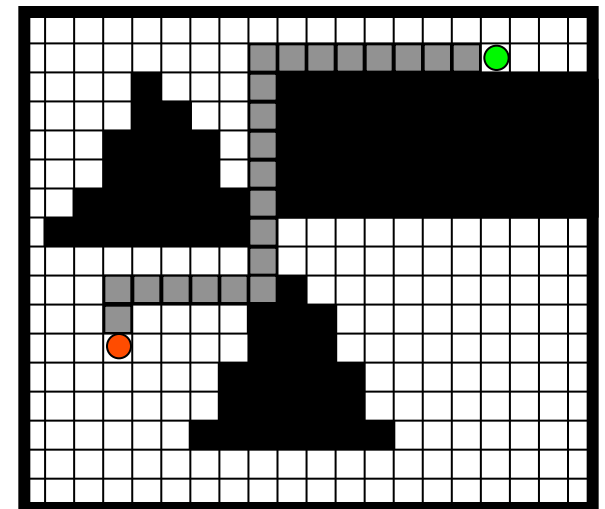
- $h(x) \leq h^*(x)$ to guarantee that A^* finds the lowest-cost path
- In this case, h is an “admissible” heuristic
- 2 commonly used ones:

- Manhattan distance $d_1(\mathbf{p}, \mathbf{q}) = \|\mathbf{p} - \mathbf{q}\|_1 = \sum_{i=1}^n |p_i - q_i|,$
- Euclidean distance

$$d(\mathbf{p}, \mathbf{q}) = \sqrt{(p_1 - q_1)^2 + (p_2 - q_2)^2 + \cdots + (p_n - q_n)^2} = \sqrt{\sum_{i=1}^n (p_i - q_i)^2}.$$

Once the search is done, and we have found the goal

- We have a tree that contains a path from the start (root) to the goal (some leaf)
- Follow the parent pointers in the tree and trace back from the goal to the root, keeping track of which states you pass through
- This set of states constitutes your plan
- To execute the plan, use your controller to face the first state in the plan, and then drive to it
- Once at the state, face and drive to the next state





Progression vs. Regression

- **Progression (forward-chaining):**
 - Choose action whose preconditions are satisfied
 - Continue until goal state is reached
- **Regression (backward-chaining):**
 - Choose action that has an effect that matches an unachieved subgoal
 - Add unachieved preconditions to set of subgoals
 - Continue until set of unachieved subgoals is empty
- Progression: + Simple algorithm (“forward simulation”)
 - Often large branching factor
- Regression: + Focused on achieving goals
 - Need to reason about actions
 - ***Regression is incomplete, if it is not possible to determine the set of all possible predecessor states***



Design Choices

- How is your map described? This may have an impact on the state space for your planner
 - Is it a grid map?
 - Is it a list of polygons?
 - The critical choice for motion planning is state space
 - The other choices tend to affect computational performance, not robot performance
- What kind of controller do you have?
 - Do you just have controllers on distance and orientation?
 - Do you have behaviours that will let you do things like follow walls?
- What do you care about?
 - The shortest path?
 - The fastest path?
- What kind of search to use?
 - Do you have a good heuristic?
 - If so, then maybe A* is a good idea.

Evaluating Search Algorithms

- **Completeness**
 - Guarantees finding a solution whenever one exists
- **Time Complexity (generally exponential in tree depth)**
 - How long (worst or average case) does it take to find a solution? Usually measured in terms of the number of nodes expanded.
- **Space Complexity**
 - How much memory is used by the algorithm? Usually measured in terms of the maximum size that the "nodes" list becomes during the search.
- **Optimality**
 - If a solution is found, is it guaranteed to be an optimal one? That is, is it the one with minimum cost?

A*: Performance

- Complete (provided finite boundary condition and path cost greater than a positive constant)
- Optimal (in terms of path cost)
- Memory inefficient:
 - More efficient version: iterative deepening A*
- Exponential growth of search space with respect to length of solution (worst case)
- Can be polynomial if the error of the heuristic grows no faster than $O(\log(h^*(x)))$ where $h^*(x)$ is the exact distance

Pseudocode: A*

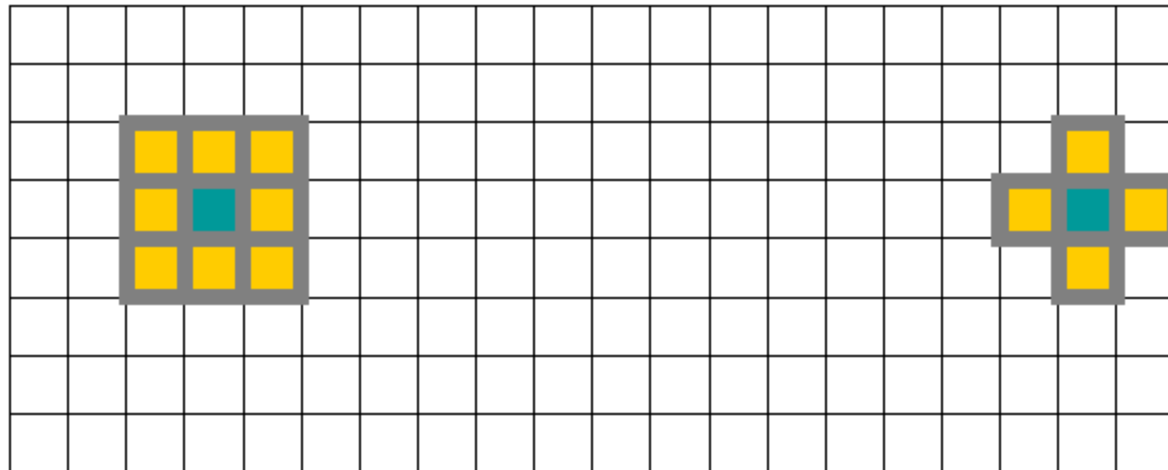
```
while the open list is not empty
  find the node with the least f on the open list, call it "q"
  pop q off the open list
  generate q's successors and set their parents to q
  for each successor
    if successor is the goal, stop the search
    successor.g = q.g + distance between successor and q
    successor.h = distance from goal to successor
    successor.f = successor.g + successor.h
    if a node with the same position as successor and a lower f is in the
    OPEN or CLOSED list
      skip this successor
    otherwise, add the node to the open list
  end
  push q on the closed list
end
```

Wavefront Planner

- Does not rely on queues and stacks
- Search order equivalent to BFS
- Uses a technique called dynamic programming:
 - Simplify a complicated problem by breaking it down into many subproblems

Connectivity

- 8-Point Connectivity
- 4-Point Connectivity



8-point connectivity will generate diagonal shortest paths which might not actually be traversable by the robot.

Wavefront Planning Setup

7	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
6	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
4	0	0	0	0	1	1	1	1	1	1	1	1	0	0	0	
3	0	0	0	0	1	1	1	1	1	1	1	1	0	0	0	
2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	2	
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Wavefront in Action (1)

Cell value = $\min(\text{non-0/1 neighbors}) + 1$

7	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
6	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
4	0	0	0	0	1	1	1	1	1	1	1	1	0	0	0	
3	0	0	0	0	1	1	1	1	1	1	1	1	0	0	0	
2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
1	0	0	0	0	0	0	0	0	0	0	0	0	0	3	3	
0	0	0	0	0	0	0	0	0	0	0	0	0	0	3	2	
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Wavefront in Action (2)

Updates expand outward from the goal area like a “wavefront”

7	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
6	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
4	0	0	0	0	1	1	1	1	1	1	1	1	0	0	0	
3	0	0	0	0	1	1	1	1	1	1	1	1	0	0	0	
2	0	0	0	0	0	0	0	0	0	0	0	0	4	4	4	
1	0	0	0	0	0	0	0	0	0	0	0	0	4	3	3	
0	0	0	0	0	0	0	0	0	0	0	0	0	4	3	2	
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Wavefront in Action (3)

7	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
6	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
4	0	0	0	0	1	1	1	1	1	1	1	1	0	0	0	0
3	0	0	0	0	1	1	1	1	1	1	1	1	5	5	5	5
2	0	0	0	0	0	0	0	0	0	0	0	0	5	4	4	4
1	0	0	0	0	0	0	0	0	0	0	0	0	5	4	3	3
0	0	0	0	0	0	0	0	0	0	0	0	0	5	4	3	2
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Wavefront in Action (4)

7	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
6	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
4	0	0	0	0	1	1	1	1	1	1	1	1	6	6	6
3	0	0	0	0	1	1	1	1	1	1	1	1	5	5	5
2	0	0	0	0	0	0	0	0	0	0	0	6	5	4	4
1	0	0	0	0	0	0	0	0	0	0	0	6	5	4	3
0	0	0	0	0	0	0	0	0	0	0	0	6	5	4	3
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

Wavefront (Complete)

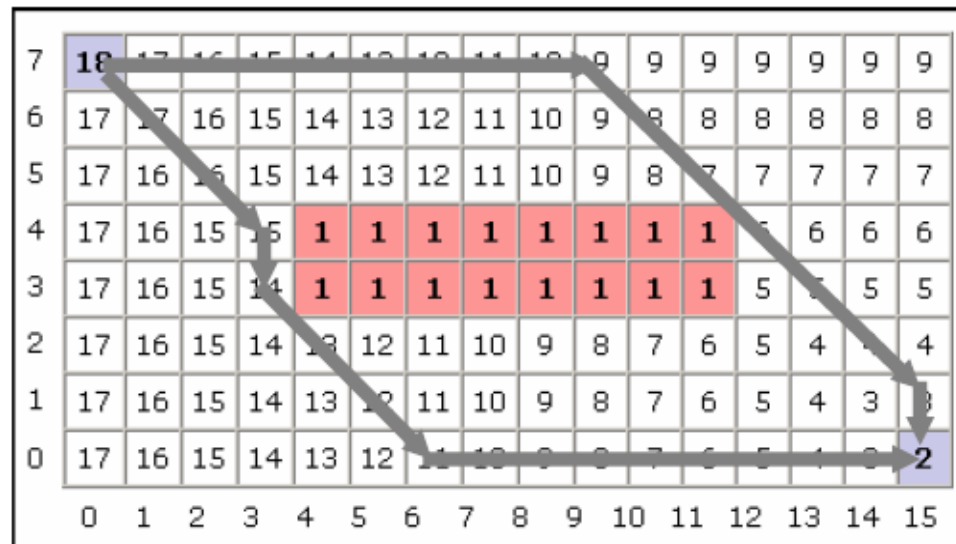
- You're done
 - Remember, 0's should only remain if unreachable regions exist

7	18	17	16	15	14	13	12	11	10	9	9	9	9	9	9	
6	17	17	16	15	14	13	12	11	10	9	8	8	8	8	8	
5	17	16	16	15	14	13	12	11	10	9	8	7	7	7	7	
4	17	16	15	15	1	1	1	1	1	1	1	1	6	6	6	
3	17	16	15	14	1	1	1	1	1	1	1	1	5	5	5	
2	17	16	15	14	13	12	11	10	9	8	7	6	5	4	4	
1	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	
0	17	16	15	14	13	12	11	10	9	8	7	6	5	4	2	
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Following the Path

- To find the shortest path, according to your metric, simply always move toward a cell with a lower number
 - The numbers generated by the Wavefront planner are roughly proportional to their distance from the goal

Two
possible
shortest
paths
shown





Wavefront Summary

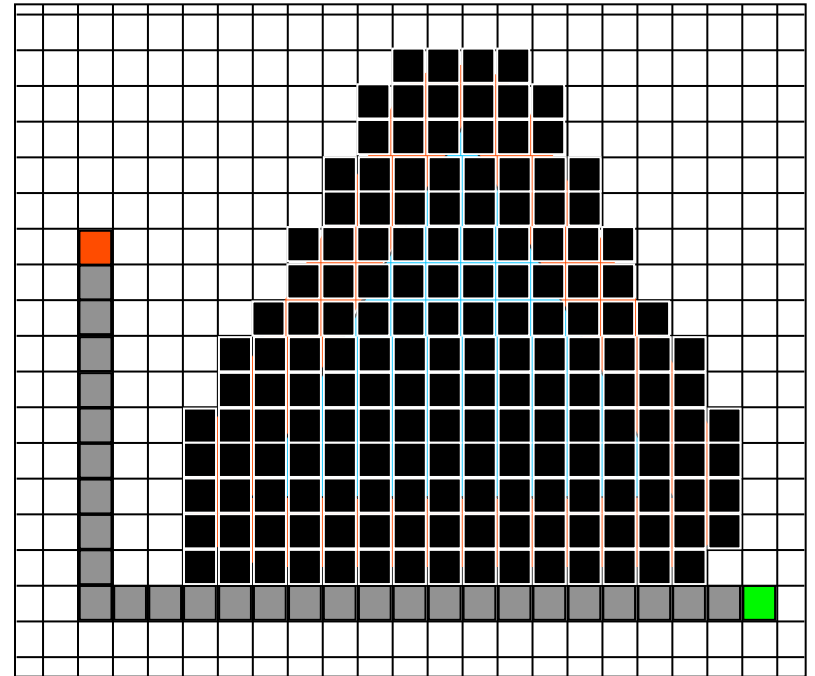
- Maintain 2 data structures (also can track current cost)
 - Old cost map
 - New cost map
- Iterate over the grid until the start square has a non-zero value
- For each cell:
 - Find lowest cost neighboring, unoccupied square and add 1 to the cost
 - If the current cost is 0 or if the new cost is lower than the current cost, annotate the cell (new cost map) with the new cost.
- Your path is defined by any uninterrupted sequence of decreasing numbers reaching the goal

Path Planning

<https://www.youtube.com/watch?v=P72CQwiwD6k&feature=youtu.be>

A problem with plans

- We have a plan that gets us from the start  to the goal .
- What happens if we take an action that causes us to leave the plan?

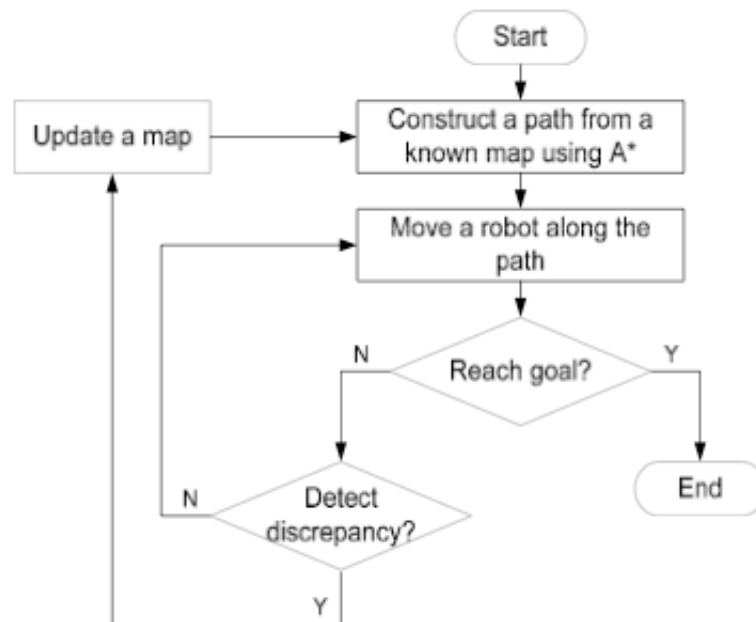


We can replan!

Re-Planning

- What happens when the world model changes?
You re-plan!
- Re-planning options
 - Plan from scratch
 - Repair existing plan
 - Often easier to plan from goal to start in this case because goal stays fixed and start node keeps moving around

A* Replanner



- Optimal
- Inefficient and impractical in expansive environments – the goal is far away from the start and little map information exists (Stentz 1994)

How can we do better in a partially known and dynamic environment?

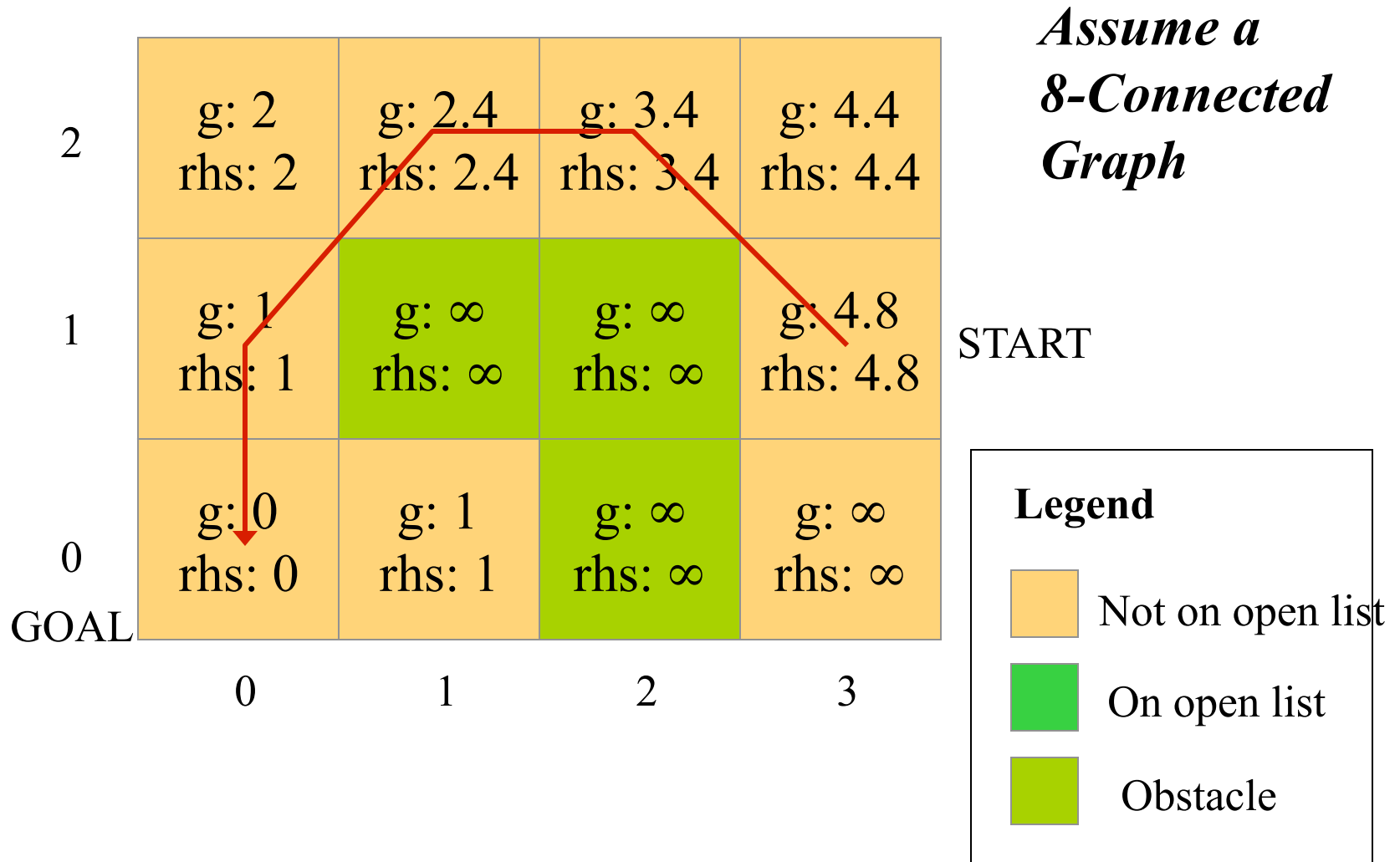
D^{*1} , D^* Lite

- Based on A^*
 - Store pending nodes in a priority queue
 - Process nodes in order of increasing objective function value
- Incrementally repair solution paths when changes occur
 - Fast re-planning
- Many variations (DD^* , Field D^* , etc.)
- Field D^* is used by the Mars rovers.

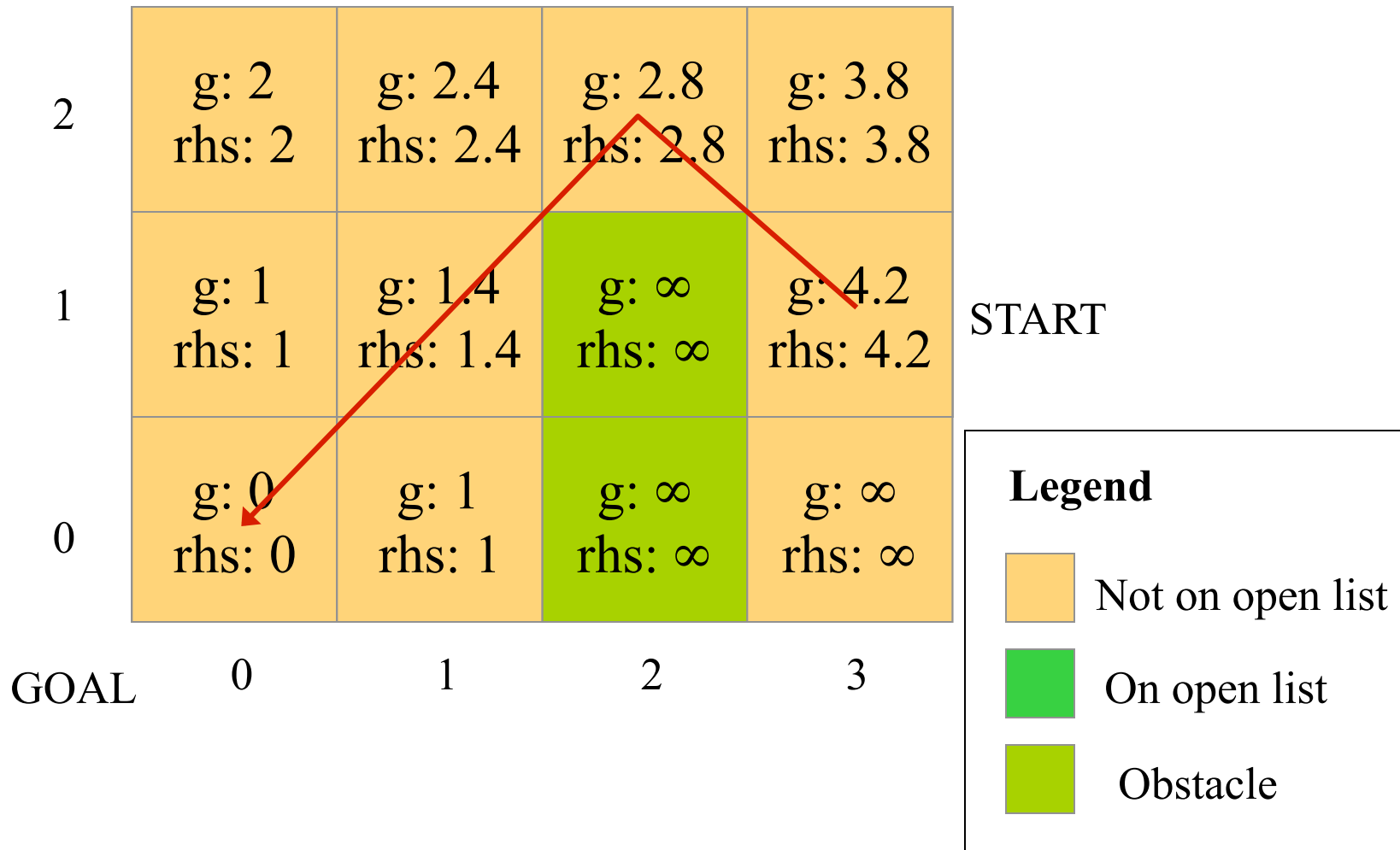
D* Lite: Functionality

- Similar functionality to D*, simpler algorithm
- Maintains two estimates of costs per node
 - g : estimate of the objective function value
 - rhs : one-step lookahead estimate of the objective function value (based on what we know)
- Defines “consistency”
 - Consistent $\Rightarrow g = rhs$
 - Inconsistent $\Rightarrow g \neq rhs$
- Inconsistent nodes go on the priority queue (“open list”) for processing
- Predecessor nodes get their rhs updated
- No back pointers

D* Lite: Planning



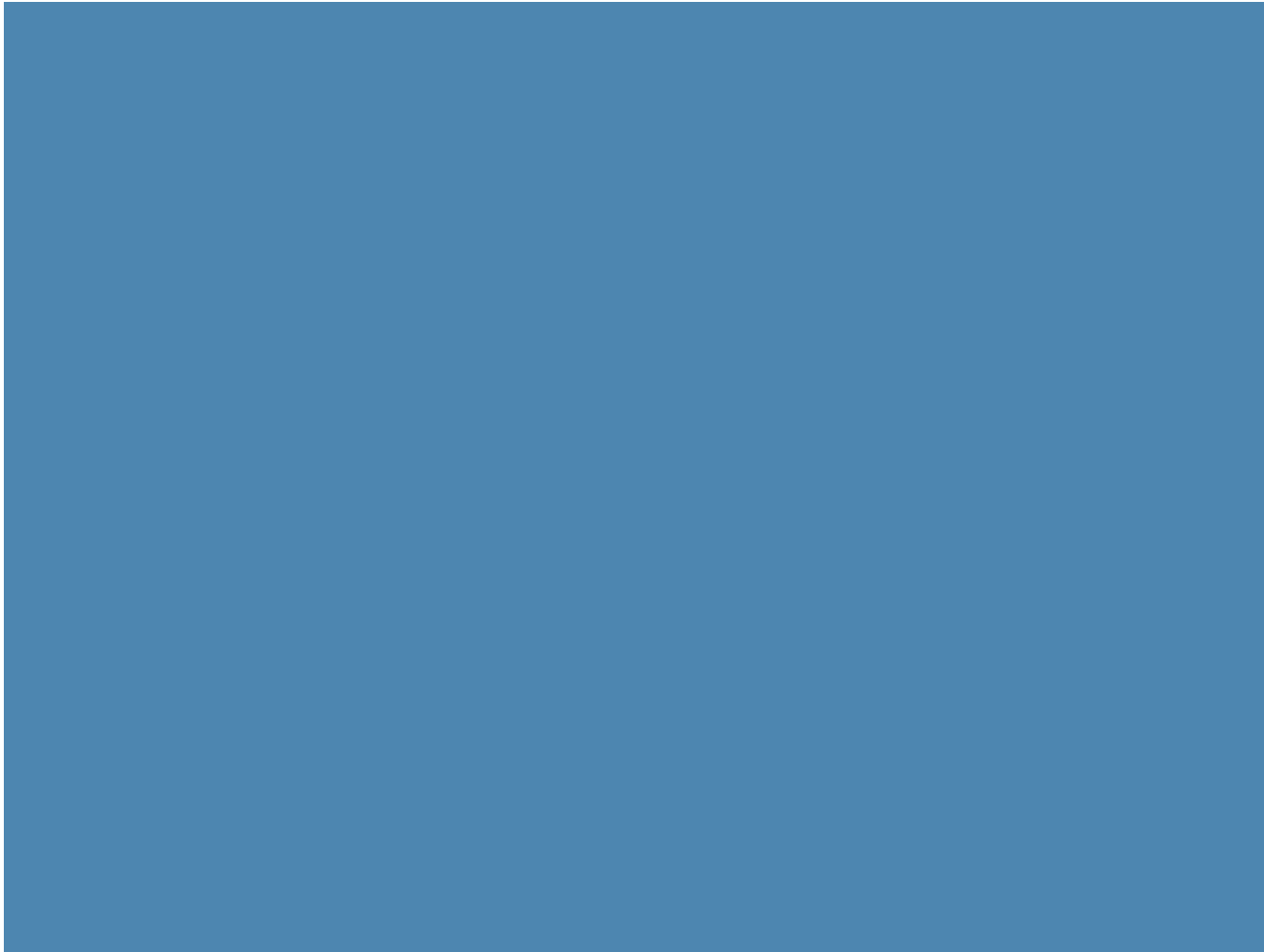
D* Lite: Replanning



Path Planning: Dynamic

<https://www.youtube.com/watch?v=DxzRYYMjxKY>

Using an A* Planner



Mission Planning

- Need to come up with high-level goals for a robot in a non-geometric space
- Less emphasis on cost and more emphasis on goal achievement
- Problems:
 - Choosing operators and representation
 - Finding the right level of granularity
 - Interleaving planning and execution

Classical Planning vs. Path Planning

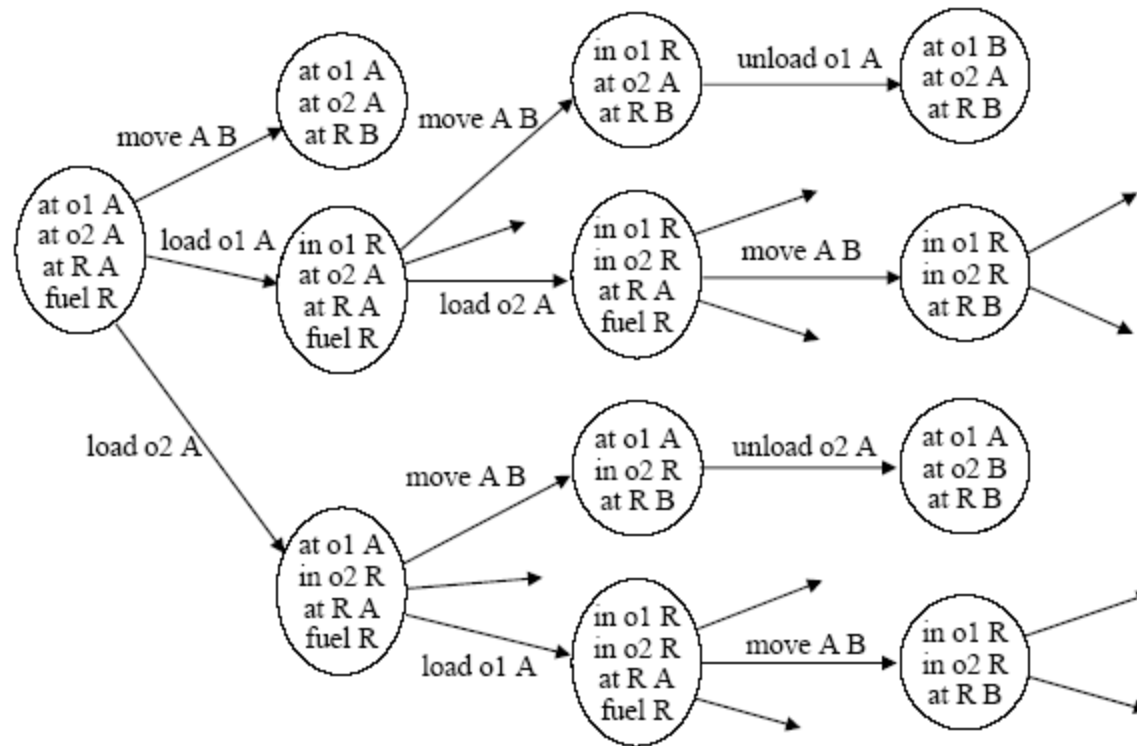
Classical

- Connectivity graph doesn't exist a priori; must be constructed during planning
- Non metric space; difficult to tell how close a node is to the goal state
- Determining completeness of search is harder

Path Planning

- Map and connectivity is usually known
- Distance metrics exist for approximating the distance between nodes
- Map is usually finite so easier to tell when search is complete

State-Space Planner

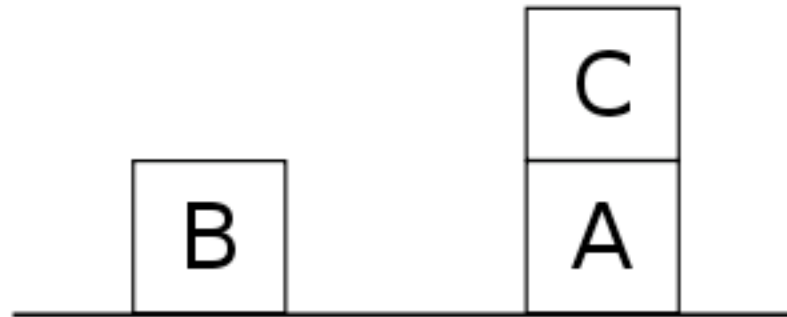


Number of states can be quite large for real-world planning problems.....

PDDL Specification

```
(define (domain blocksworld)
  (:predicates (clear ?x)
    (holding ?x)
    (on ?x ?y))
  (
    :action stack
    :parameters (?ob ?underob)
    :precondition (and (clear ?underob) (holding ?ob))
    :effect (and (holding nil) (on ?ob ?underob)
      (not (clear ?underob)) (not (holding ?ob)))
  )
)
```

Sussman Anomaly



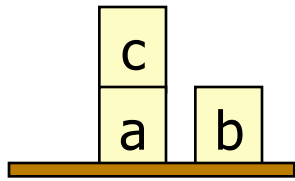
2 subgoals

- A on B
- B on C

Planner must interleave goal achievement.

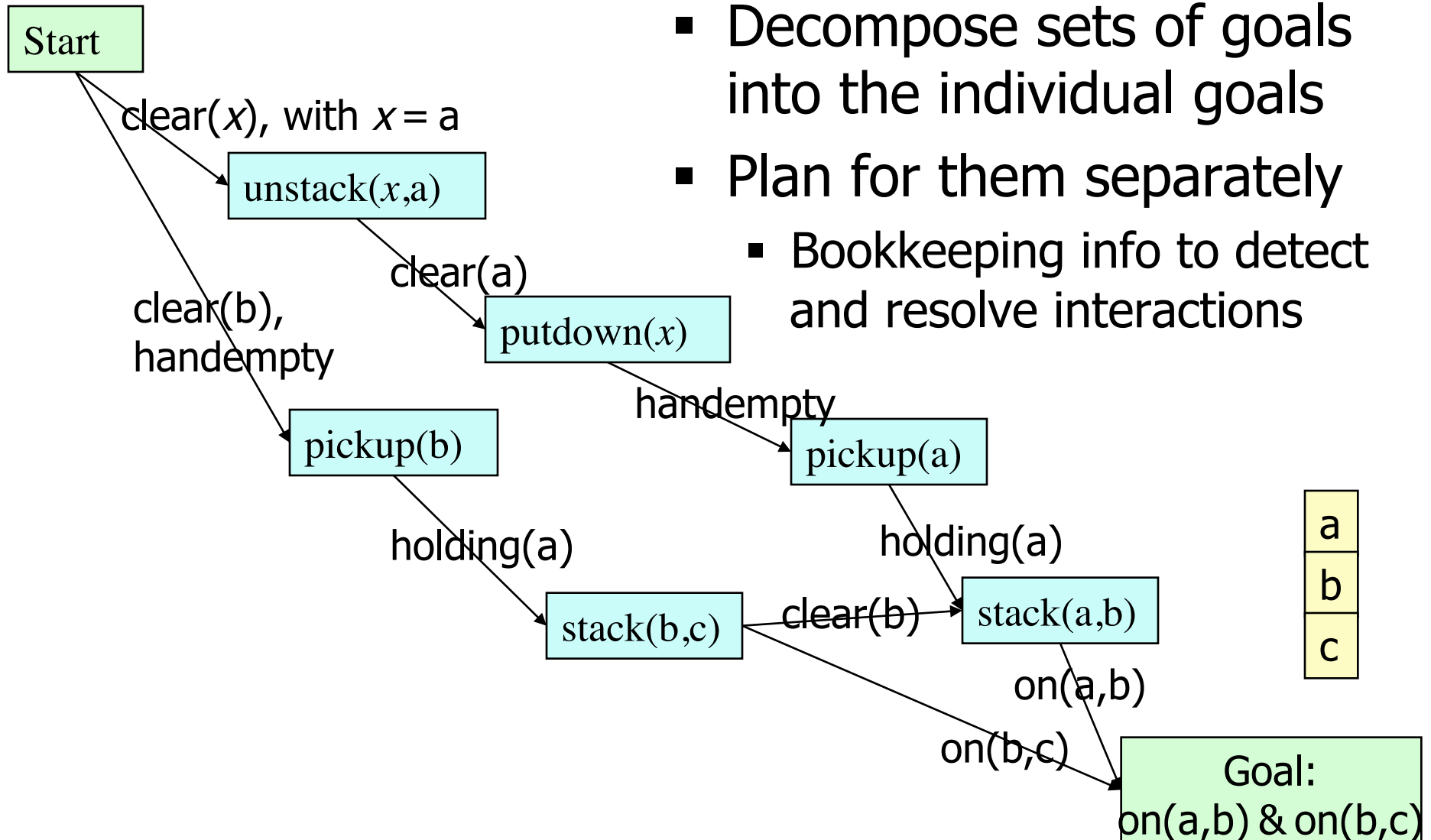
Completing either goal in entirety will render the other goal impossible

The Sussman anomaly shows that methods that treat subgoals independently are not guaranteed to be complete.



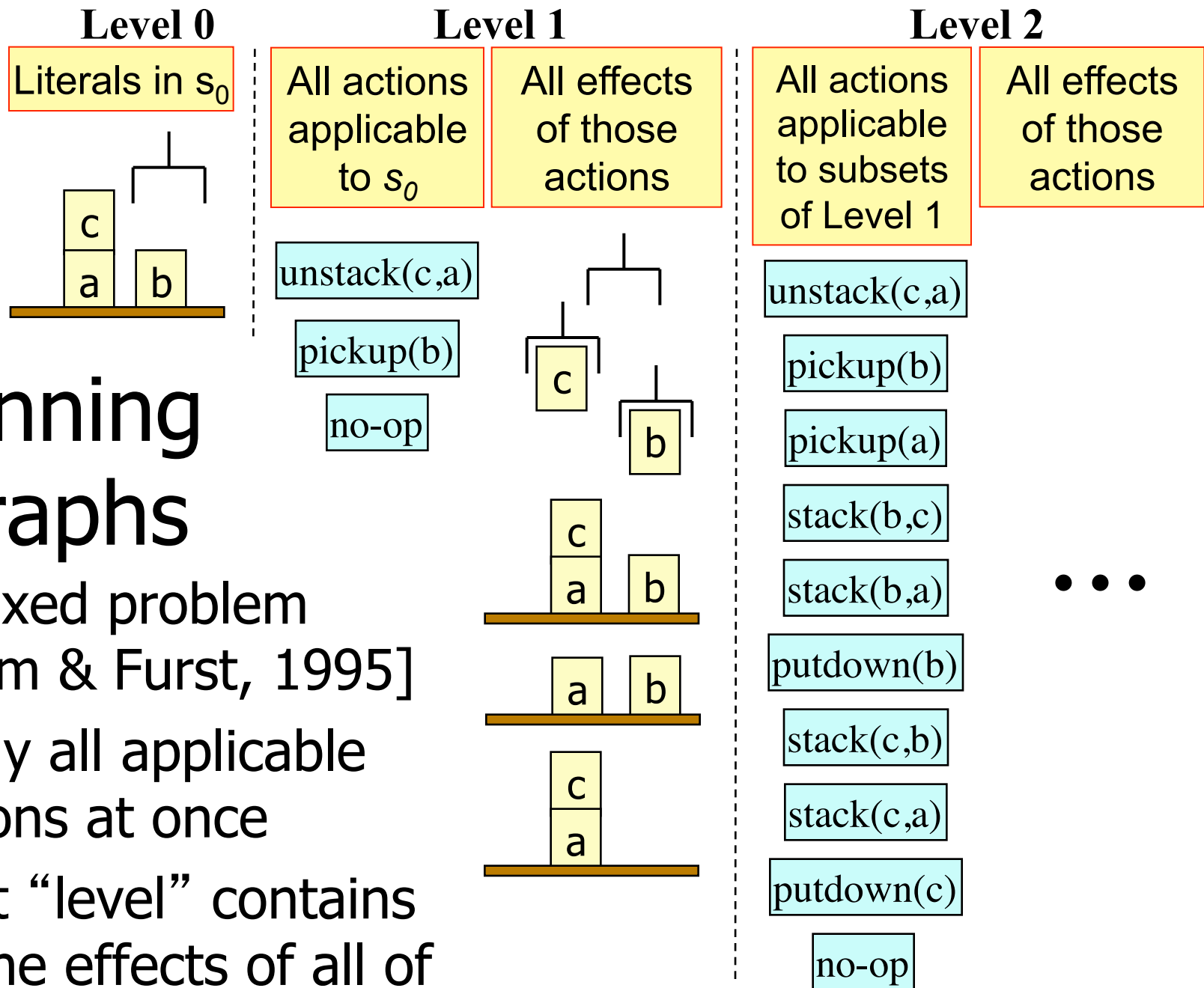
Plan-Space Planning (UCPOP)

- Decompose sets of goals into the individual goals
- Plan for them separately
 - Bookkeeping info to detect and resolve interactions



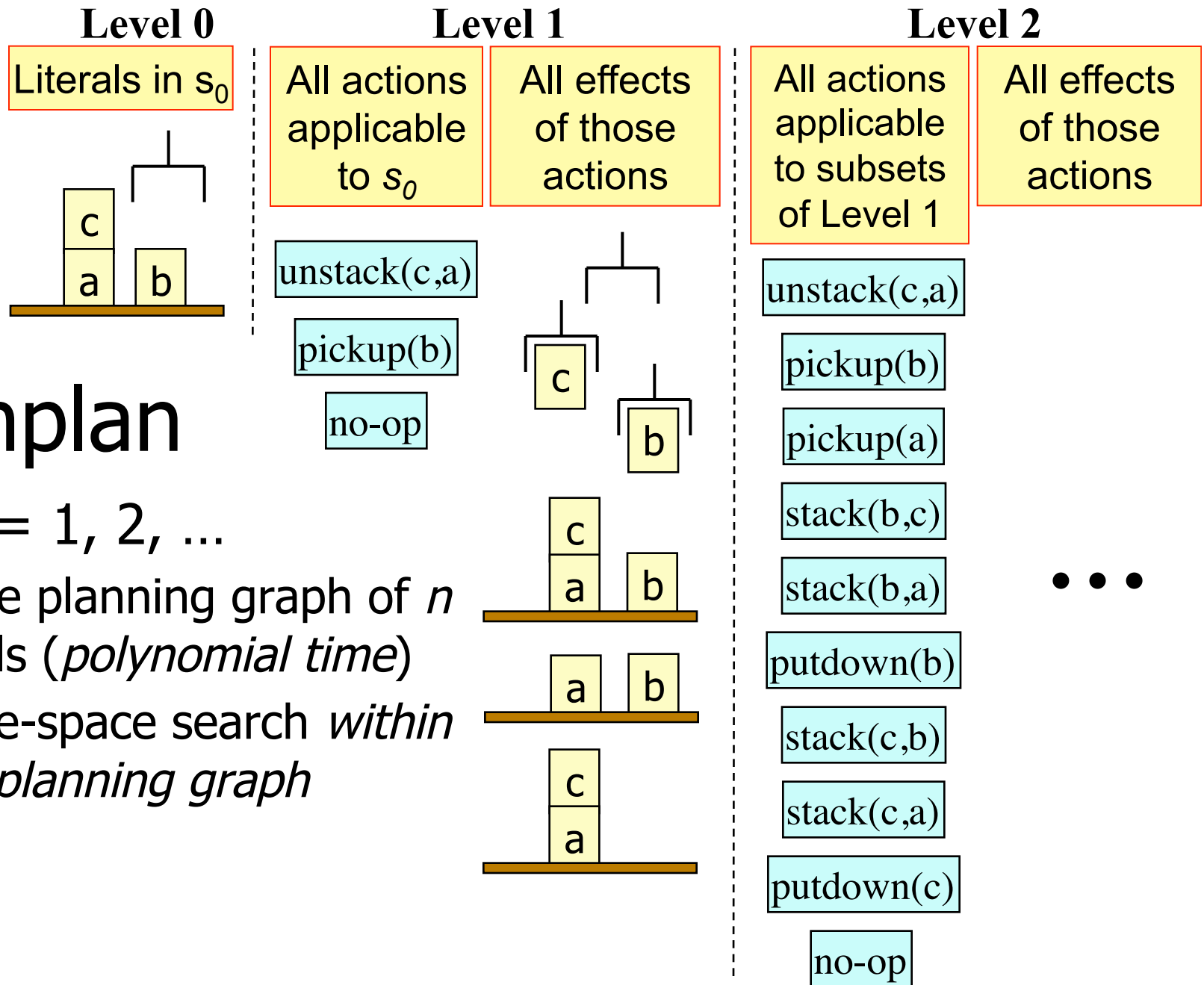
Planning Graphs

- Relaxed problem [Blum & Furst, 1995]
- Apply all applicable actions at once
- Next “level” contains all the effects of all of those actions

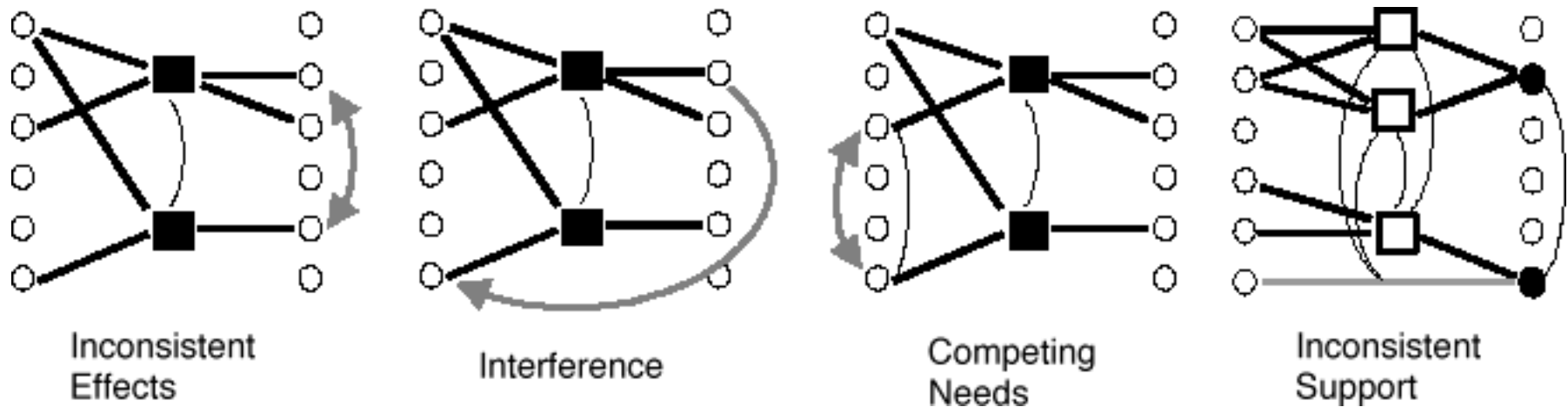


Graphplan

- For $n = 1, 2, \dots$
 - Make planning graph of n levels (*polynomial time*)
 - State-space search *within the planning graph*



Mutual Exclusion



- Two actions at the same action-level are mutex if
 - *Inconsistent effects*: an effect of one negates an effect of the other
 - *Interference*: one deletes a precondition of the other
 - *Competing needs*: **they have mutually exclusive preconditions**
- Otherwise they don't interfere with each other
 - Both may appear in a solution plan
- Two literals at the same state-level are mutex if
 - *Inconsistent support*: one is the negation of the other, **or all ways of achieving them are pairwise mutex**

Example

- Suppose you want to prepare dinner as a surprise for your sweetheart (who is asleep)

$s_0 = \{\text{garbage}, \text{cleanHands}, \text{quiet}\}$

$g = \{\text{dinner}, \text{present}, \neg\text{garbage}\}$

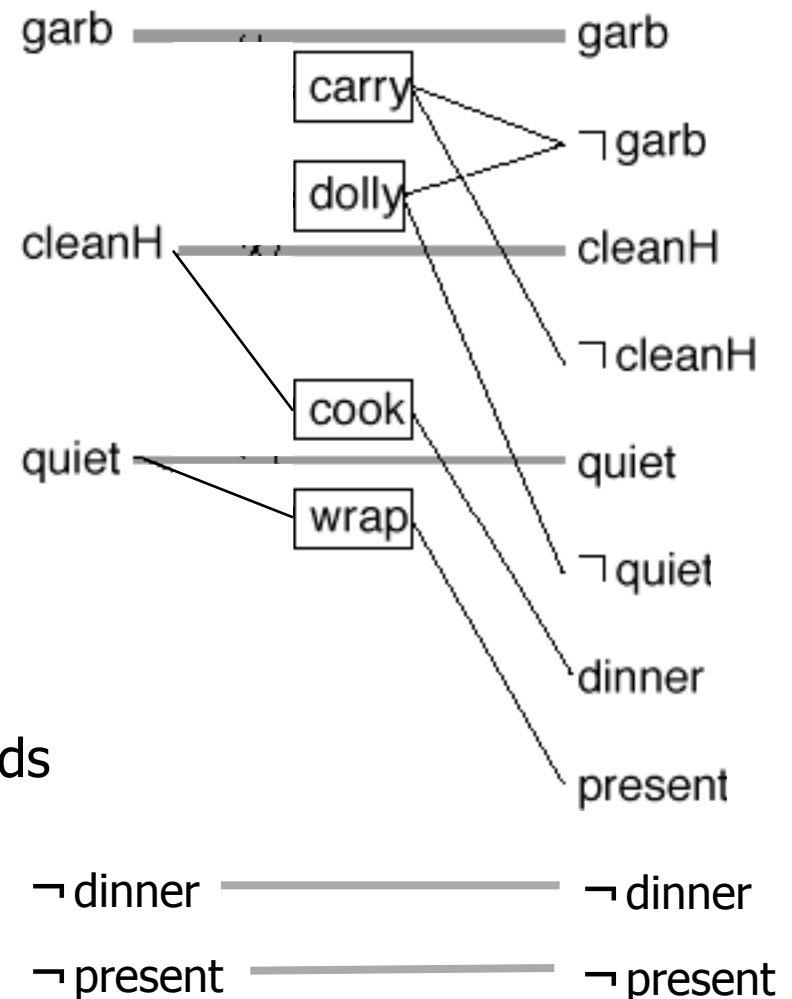
<u>Action</u>	<u>Preconditions</u>	<u>Effects</u>
cook()	cleanHands	dinner
wrap()	quiet	present
carry()	<i>none</i>	$\neg\text{garbage}, \neg\text{cleanHands}$
dolly()	<i>none</i>	$\neg\text{garbage}, \neg\text{quiet}$

Also have the maintenance actions: one for each literal

Example (continued)

- state-level 0:
 $\{\text{all atoms in } s_0\} \cup$
 $\{\text{negations of all atoms not in } s_0\}$
- action-level 1:
 $\{\text{all actions whose preconditions}$
 $\text{are satisfied and non-mutex in } s_0\}$
- state-level 1:
 $\{\text{all effects of all of the}$
 $\text{actions in action-level 1}\}$

state-level 0	action-level 1	state-level 1
---------------	----------------	---------------



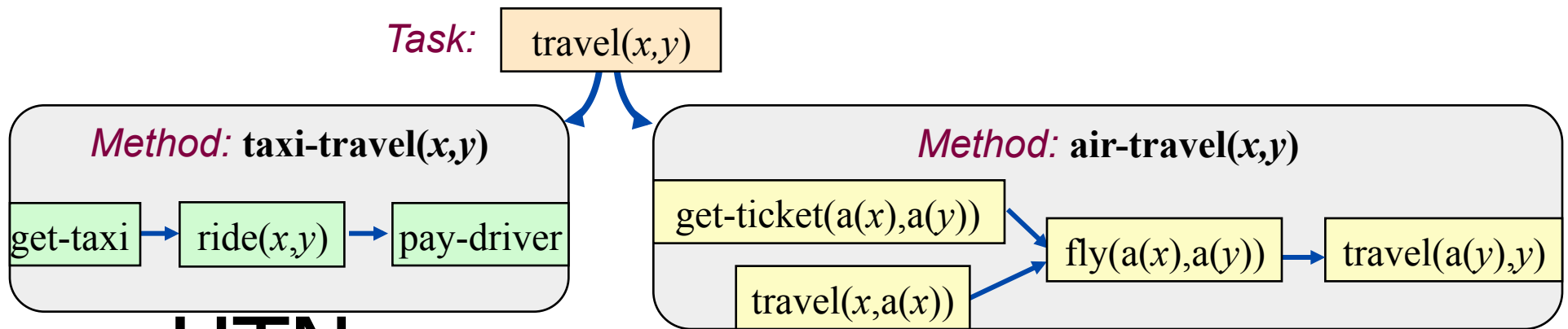
Action Preconditions Effects

cook()	cleanHands	dinner
wrap()	quiet	present
carry()	<i>none</i>	\neg garbage, \neg cleanHands
dolly()	<i>none</i>	\neg garbage, \neg quiet

Also have the maintenance actions

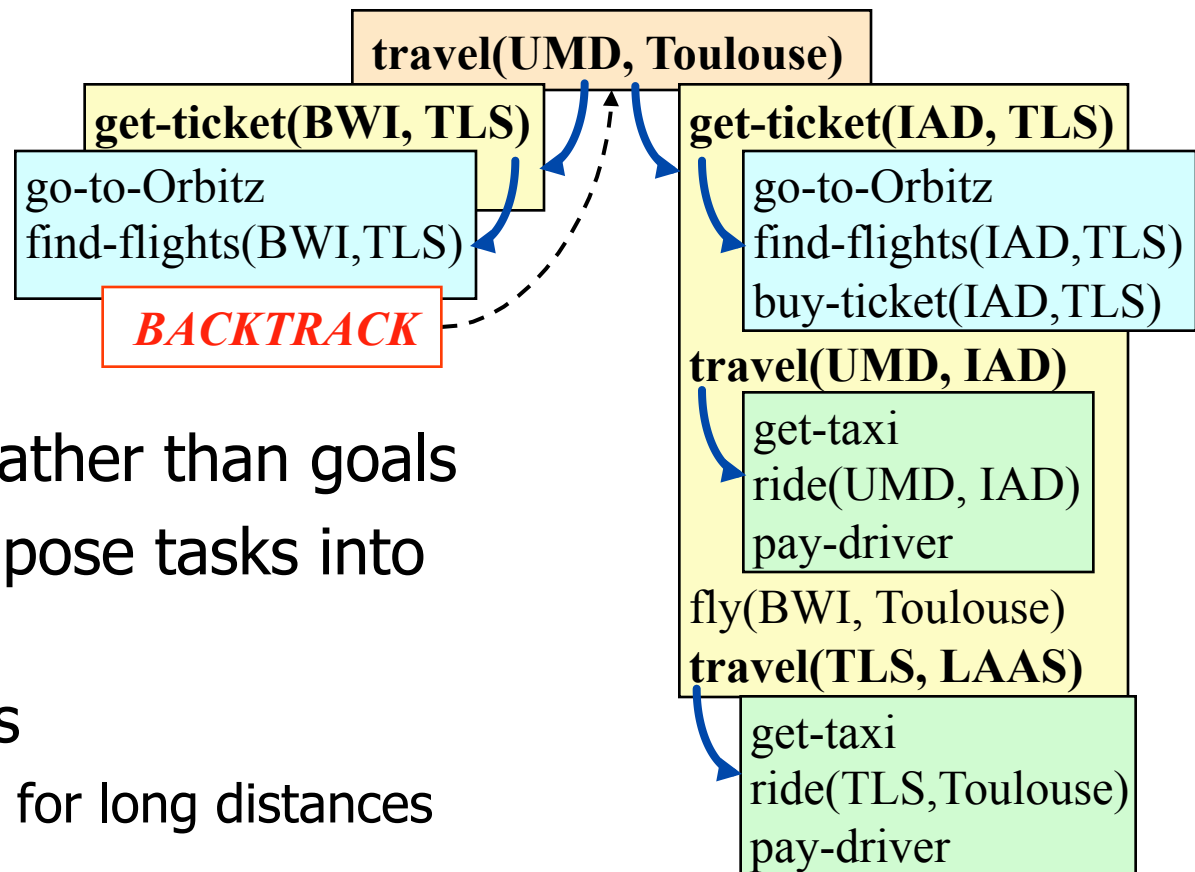
Planning as Humans Do It

- Make a plan to get you from your bed to UCF in the morning
 - Do you consider all possible options that you can take ? OR
 - Do you remember recipes and procedures that have worked successfully for you in the past?
- That' s what HTN (hierarchical-task network) planning is about!



HTN Planning

- Problem reduction
 - *Tasks* (activities) rather than goals
 - *Methods* to decompose tasks into subtasks
 - Enforce constraints
 - E.g., taxi not good for long distances
 - Backtrack if necessary



Summary

- Planning involves a lot of bookkeeping; the more that you are able to remember and use from earlier searches the better