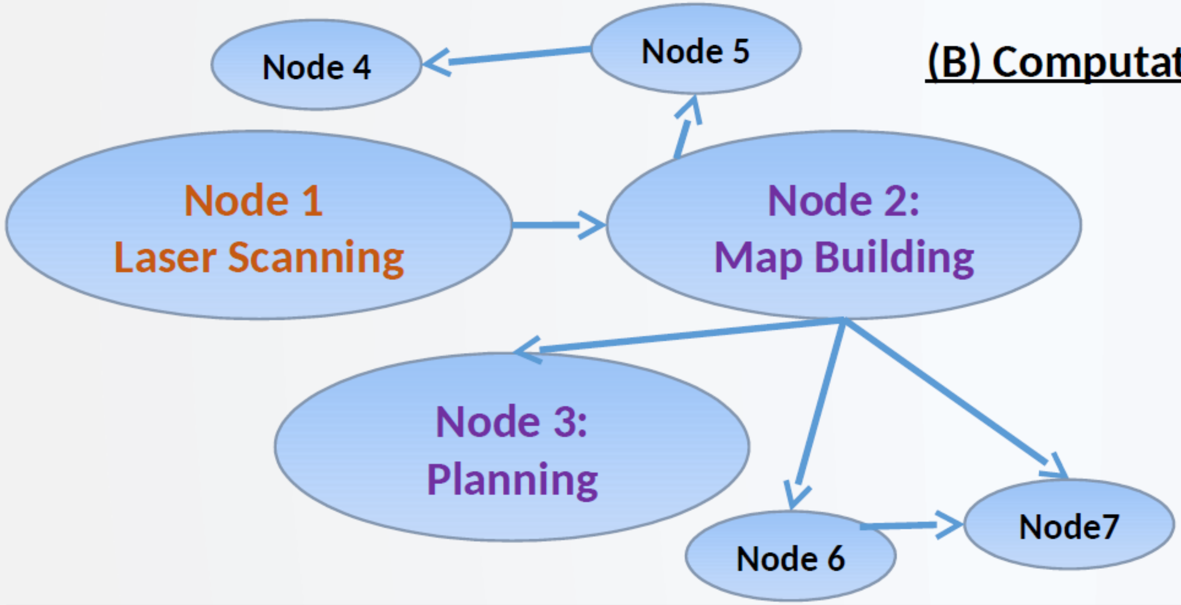
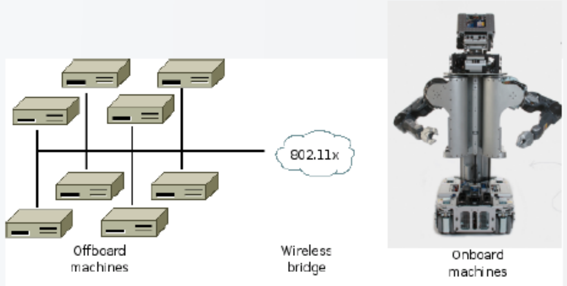
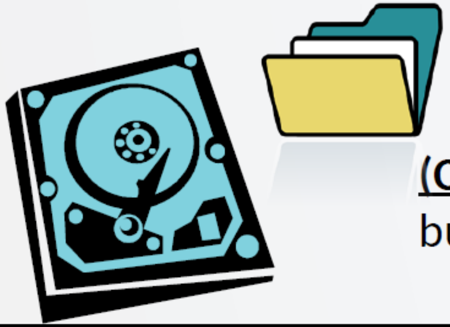
Conceptual levels of design

(A) ROS Community: ROS Distributions, Repositories



(B) Computation Graph: Peer-to-Peer Network of ROS nodes (processes).





(C) File-system level: ROS Tools for managing source code, build instructions, and message definitions.

Another View of ROS

Plumbing

- Device Drivers
- Inter-Node Communication
- Process Management

Tools

- Visualization
- Simulation
- Debugging
- User Interface

Robot Capabilities & Functions

- Robot Control
- Motion Planning
- Mapping
- Localization
- Perception
- Manipulation

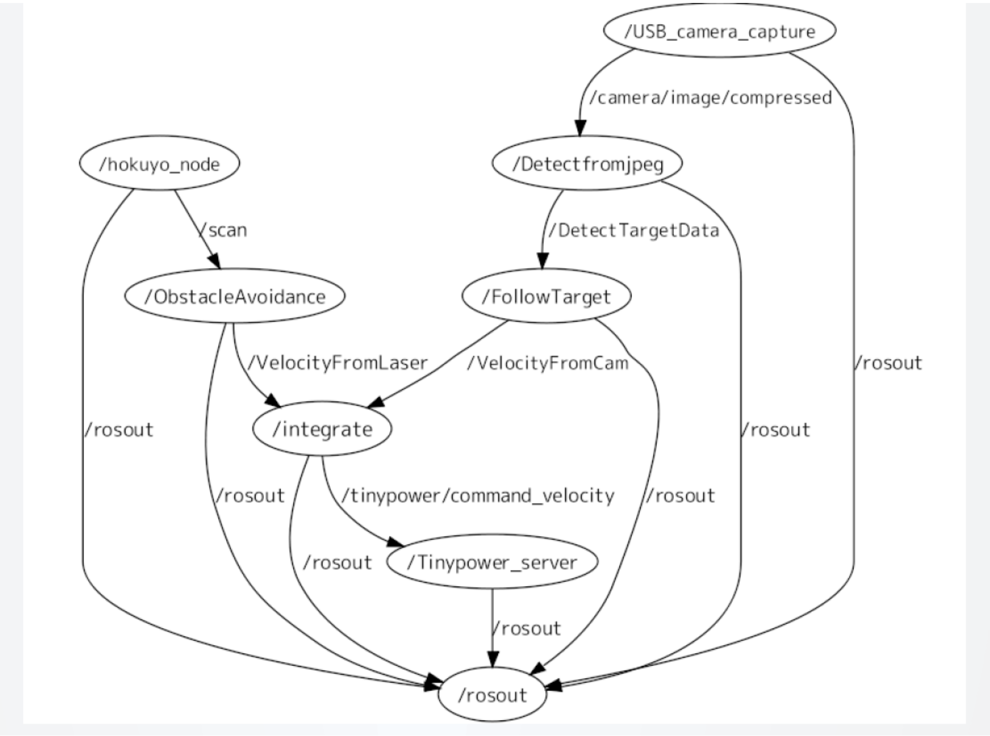
Community EcoSystem

- Package Organization
- Repositories
- Tutorials
- Documentation
- FAQ/Forum
- Workshops/Training

Many ROS Tools

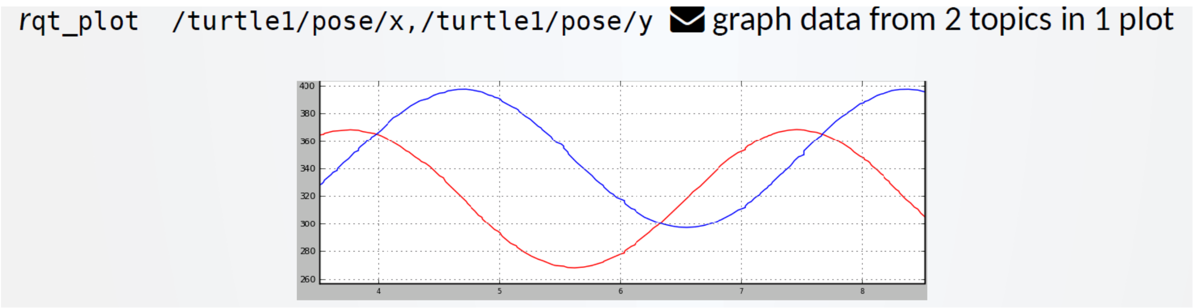
Developer Tools:

- Building ROS nodes: catkin_make
- Running ROS nodes: rosrun, roslaunch
- Viewing network topology: rqt_graph



Debugging Tools:

- **Rostopic:** display info about active topics (publishers, subscribers, data rates and content)
- rostopic echo [topic name]   (*prints topic data)*
- rostopic list *(prints list of active topics)*
- **Rqt_plot:** plots topic data

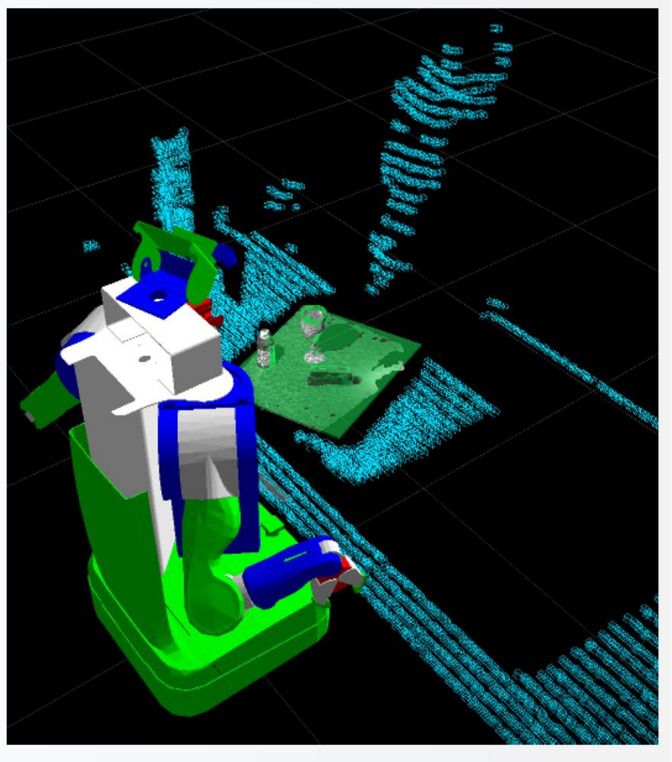


- Data logging:
  - Rosbag record [topics] –o < output_file>
- Data playback:
  - Rosbag play <input_file> --clock

Many ROS Tools

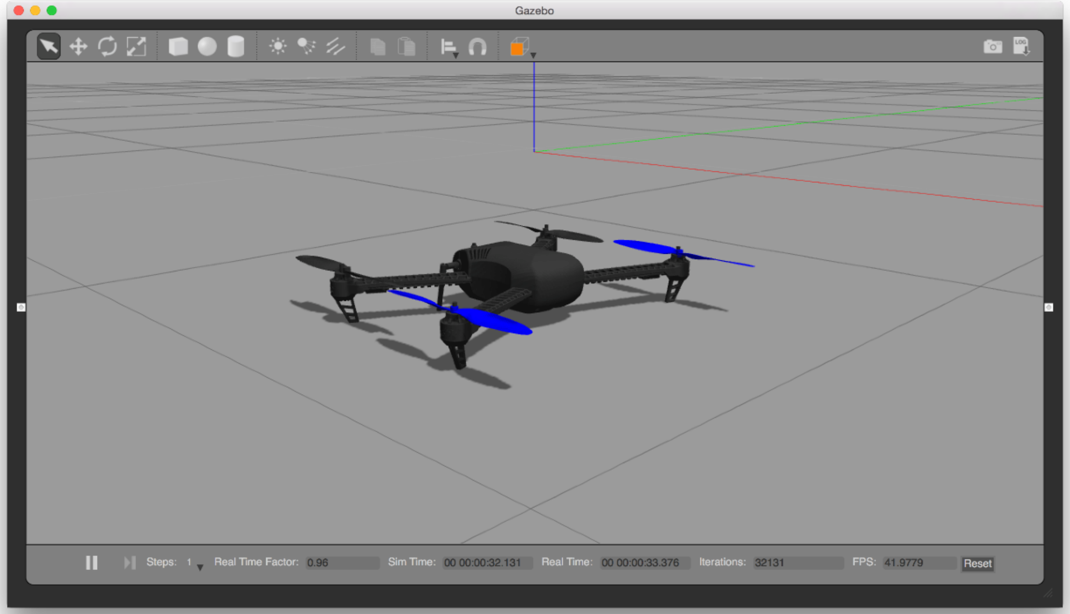
Visualization Tools: RVIZ

- Sensor and robot state data
- Coordinate frames
- Maps, built or in process
- Visual 3D debugging markers



Simulation Tools:

- **Gazebo:** started as grad student project at USC
- Can model and simulate motions/dynamics of different robots
- Can simulate sensory views
- Can build different environments
- Can run simulation from ROS code for testing



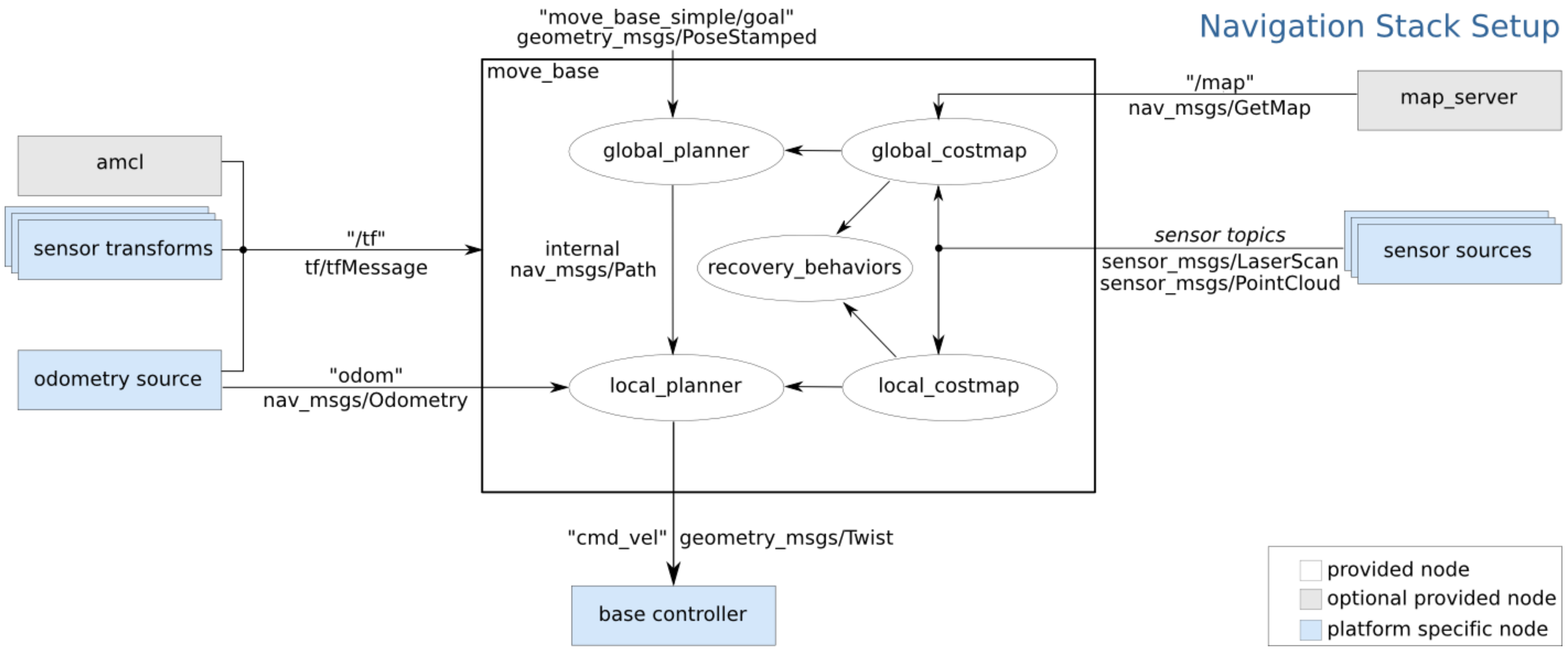
A first look at ***move_base***

move_base is a *package* that implements an *action* in ROS.

- An action can be *preempted*
- An action can provide periodic feedback on its execution

move_base is a node that *moves* a robot (the “*base*”) to a goal

- It links a *global* and *local* planner with sensory data and maps that are being built, so that the *navigation stack* can guide the robot to a goal, and have *recovery strategies*



Goals for Next Week

Download ROS distribution.

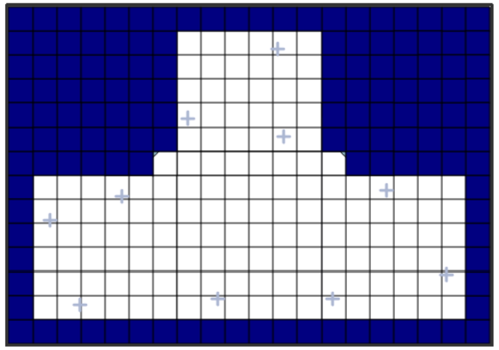
- Choose how you want to manage Ubuntu on your machine:
  - Dual boot
  - Virtual machine: (one option is the free *virtual box:* https://itsfoss.com/install-linux-in-virtualbox/)
  - Try the Windows installation?
- Install ROS (melodic is best, but kinetic might be okay)

GO through the first 2-3 steps of the *Core ROS Tutorial* at the beginner's level.

- You may prefer to start the first few steps of "*A Guided Journey to the Use of ROS*"

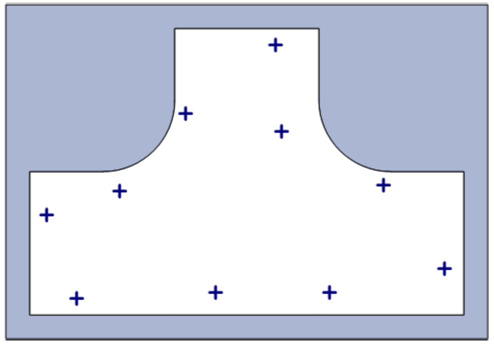
Three Major Map Models

Grid-Based:
Collection of discretized obstacle/free-space pixels



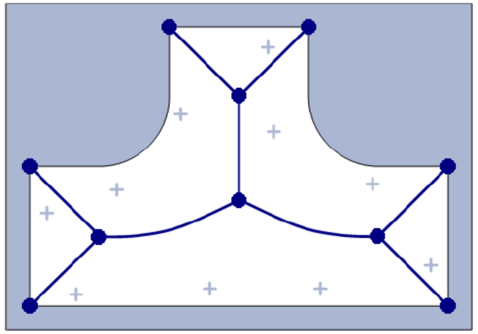
Elfes, Moravec, Thrun, Burgard, Fox, Simmons, Koenig, Konolige, etc.

Feature-Based:
Collection of landmark locations and correlated uncertainty



Smith/Self/Cheeseman, Durrant–Whyte, Leonard, Nebot, Christensen, etc.

Topological:
Collection of nodes and their interconnections

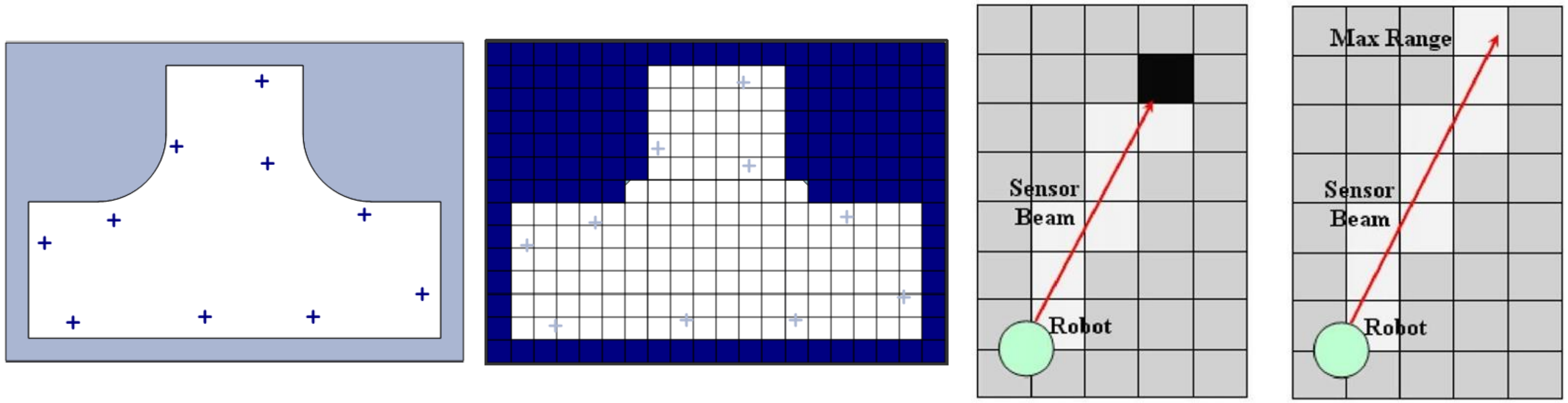


Kuipers/Byun, Chong/Kleeman, Dudek, Choset, Howard, Mataric, etc.

Gmapping

Occupancy Grid: "map" is a grid of "cells": $\{x^m_{i,j}\}$

- $x^m_{i,j} = 0$ if cell (i,j) is empty; $x^m_{i,j} = 1$ if cell (i,j) is occupied
- $p\left(x^r_{k+1}, \{\mathrm{x}^{\mathrm{m}}_{\mathrm{i,j}}\}_{k+1} \middle| \mathrm{x}^{\mathrm{r}}_{1:k}, \{\mathrm{x}^{\mathrm{m}}_{\mathrm{i,j}}\}_{k}, \mathrm{y}_{1:k+1}\right)$ (estimate cell occupancy probability)



(a) (b)

Gmapping:

- Uses a *Rao-Blackwellized* particle filter for estimator
- Actually computes $p\left(x^r_{1:T}, \{x^m_{i,j}\} \middle| \mathrm{x}^{\mathrm{r}}_{1:k}, \mathrm{x}^{\mathrm{m}}_{\mathrm{k}}, \mathrm{y}_{1:k+1}\right)$

Control & Planning for MDPs POMDPS

Autonomy (a self-governing system):

- Make Decisions and Plans, in the presence of uncertainty
  - Process and measurement noise
  - Incomplete models
  - Incomplete information
  - Adversarial conditions
- With little or no human guidance

Some key issues

- Where am I? $\Rightarrow$ SLAM
- Action selection
  - Control in Markov Decision Processes (MDPs) and POMPDs
- Planning
- Supervisory Control

Feedback Control/Action Selection

Given $x_{k+1} = f(x_k, u_k) + \eta_k$:

- State Feedback (assumes that all states are “observable”):
  - $u_k = g(x_1, x_2, \ldots, x_k, u_1, \ldots, u_{k-1})$
- Output Feedback: $y_k = h(x_k) + \omega\_k$
  - $u_k = q(y_1, \ldots, y_k, u_1, \ldots, u_{k-1})$

Feedback Aims:

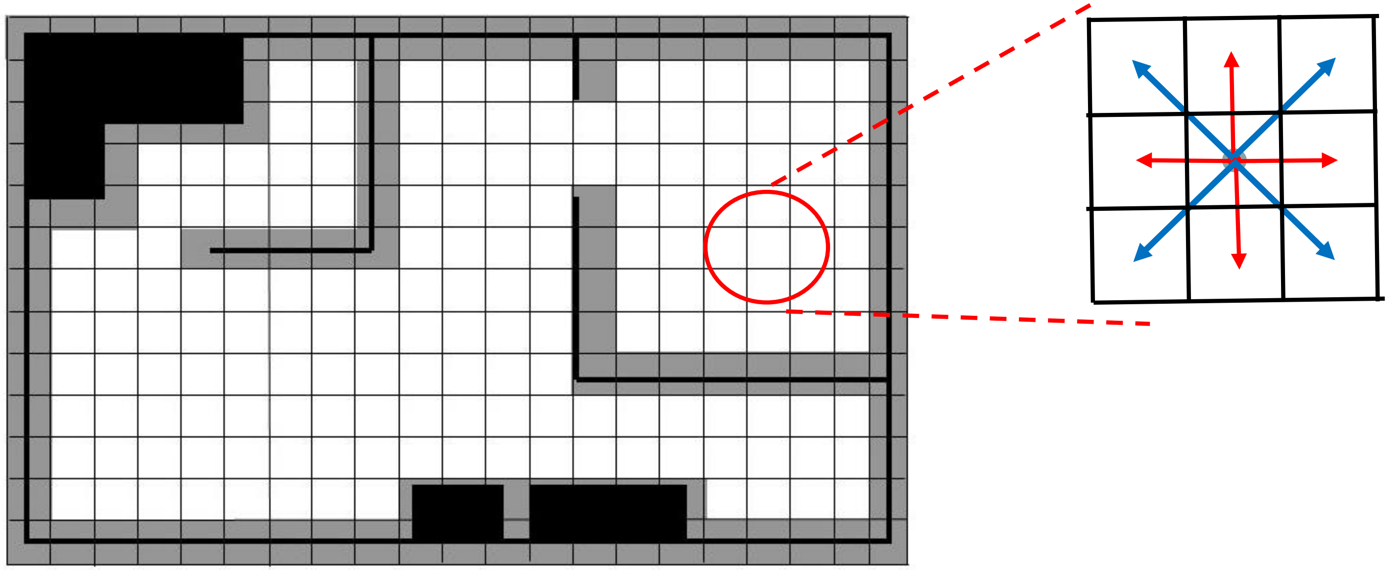
- Given a goal, maximize probability of attaining goal
- If possible, optimize other criteria while achieving goal
  - Minimize energy use, or time to goal)
- Avoid problems
  - Avoid obstacles, stay away from difficult to traverse or dangerous areas

Markov Decision Processes (MDPs)

Motivation: a model for many (but not all) dynamical systems that are part of a decision problem

Definition: A *Mark Decision Process* (MDP) consists of

- A discrete set of states, $S = \{x_1, x_2, \ldots, x_N\}$
- A set of possible actions to take in each state: $U = \{u_1, \ldots, u_k\}$
  - Set of actions can be state dependent: $U_i = U(x_i)$



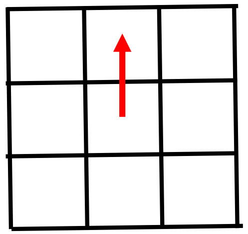
Markov Decision Processes (MDPs)

Definition (continued)**:** A *Mark Decision Process* (MDP) consists of

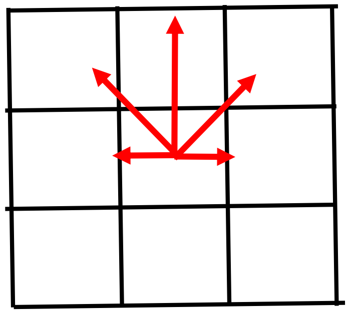
– A *transition function*, T, that describes the system "dynamics"

- Deterministic: $T: S \times U \rightarrow S$
- Stochastic: $T: S \times U \rightarrow Prob(S)$.
  – I.e., a probability distribution over the next states, condition and the current state and action: $p(x'|x, u)$

Deterministic:



Stochastic: Probability proportional to length of arrow



- The *Markov Assumption* holds:
  – $p(x_{k+1}|x_0, x_1, \dots, x_k, u_0, \dots, u_k) = p(x_{k+1}|x_k, u_k)$
  – the prediction of state x_{k+1} only depends upon x_k, u_k, and not prior states and controls
  – Future system states only depend upon the current state (and control), and not on the prior history → *memoryless*

Markov Decision Processes (MDPs)

Definition (continued)**:** A *Mark Decision Process* (MDP) consists of

– A *reward function* $r(x,u) \rightarrow \mathbb{R}$

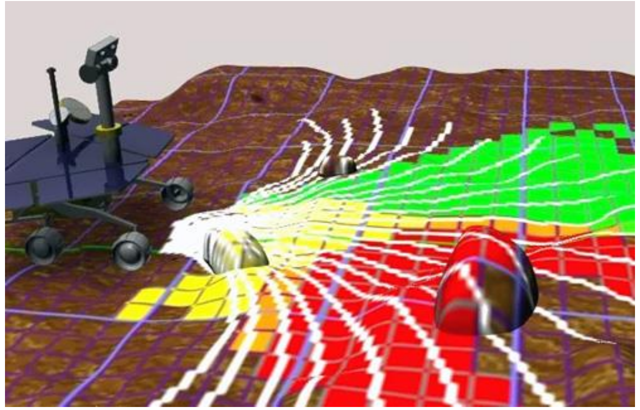
- Reward can incorporate *goal information*

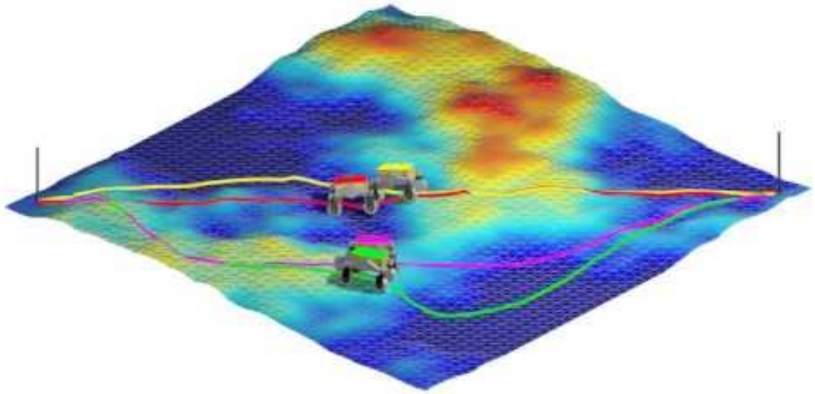
$$r(x,u) = \begin{cases} +100 & if\ u\ leads\ to\ the\ goal \\ -1 & otherwise \end{cases}$$

- Reward can incorporate costs:

$r(x,u)$= amount of energy to execute action u

$r(x,u)$ = penalty to be in state x (e.g., traversibility analysis)





Policy

Definition: A *Control Policy,* or *Policy*, prescribes an *action* or *control*

- $u_k = \pi(x_k)$ for a fully observable system (MDP)
- $u_k = \pi(y_{1:k}, u_{1:k-1})$ for partially observable system (more later)
- Policy π can be deterministic or stochastic
  - Deterministic: $u = \pi(x)$
  - Stochastic: $\pi(u|x) = Prob[u_t = u | s_t = x]$

We want to find a policy that

- Realizes the goal as best as possible
- Considers constraints
- Considers the costs of its actions

Approach: Find $\pi(x)$ that *maximizes* a cumulative reward

Cumulative Reward

$$R_T = E\left[\sum_{i=0}^{T-1} \gamma^i \ r(x_i, u_i)\right] \qquad R_T^{\pi} = E\left[\sum_{i=0}^{T-1} \gamma^i \ r(x_i, u_i) | u = \pi(x)\right]$$

T is the ***horizon***

- $T = 1$: *"Greedy"*
- T is finite: *"Finite-Horizon Problem"*
- $T = \infty$: *"Infinite-Horizon Problem" (often used when T large)*

γ is a *discount factor:* $\gamma \in [0,1]$ or discount rate.

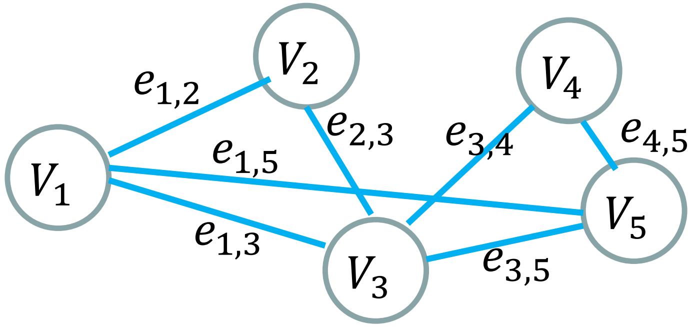
- A reward n steps away is discounted by γ^n
- Models mortality or impatience: you may die soon
- Models the preference for shorter solutions
- Needed for infinite horizon cumulative reward to be finite

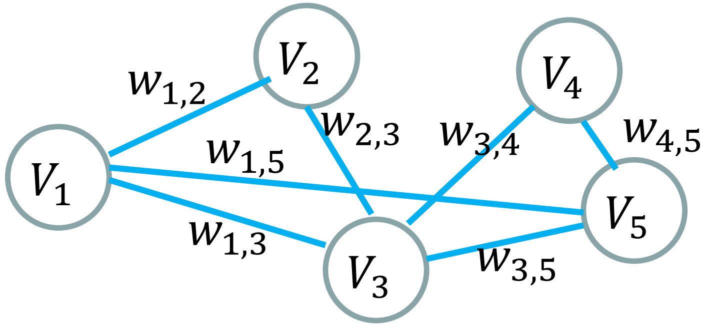
$$|R_\infty| \leq r_{max} + \gamma^1 r_{max} + \gamma^2 r_{max} + \cdots = \frac{r_{max}}{1-\gamma}; \qquad r_{max} = \max_{x,u} |r(x,u)|$$

Dynamic Programming

Let's first consider a class of problems where the system dynamics are not important

- the transitions between states are the only costs that matter.
- Said differently, the *decision* made at each state incurs a cost
- Such problem can be modeled by a graph, G=(V,E) with *weighted edges.* I.e., weight $w_{i,j}$ is associated to edge, $e_{i,j}$





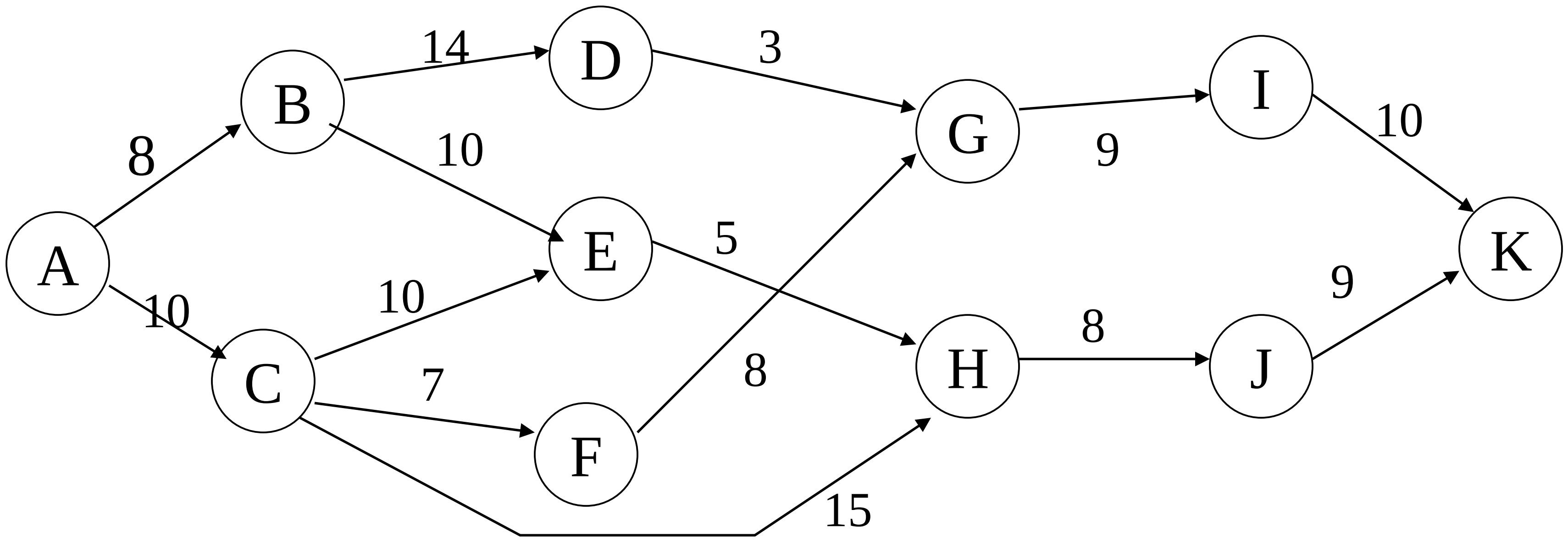
- These problems reduce down to a *shortest path* problem

Dynamic programming **(DP)** is a general optimization technique to solve these *sequential decision* problems..

It is based on the "**principle of optimality**"

Illustration of DP by shortest path problem

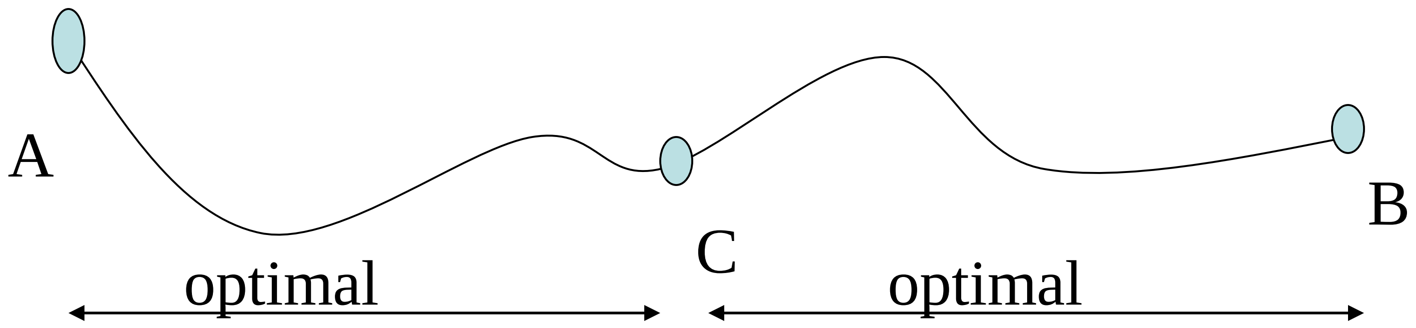
- **Problem :** We plan to construct a highway from city A to city K. Different construction alternatives and their costs are given in the following graph. Determine the highway route with the minimum total cost.



BELLMAN's principle of optimality

Basic Idea:

- if node C belongs to an optimal path from node A to node B, then the sub-path from A to C and from C to B are *also* optimal
- Any sub-path of an optimal path is optimal



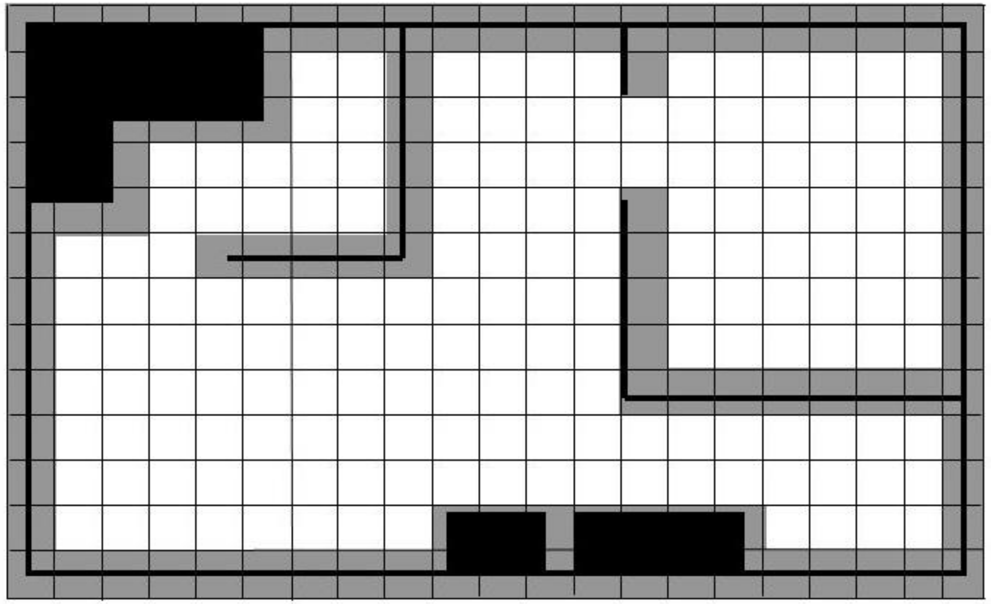
Corollary :

$$\mathbf{SP(x, y) = \min \{SP(x, z) + l(z, y) \mid z : \text{predecessor of } y\}}$$

Application to Autonomous Planning

Approximate Cellular Decomposition:

- Divide environment (or c-space) into “cells”
  - Simple shape
  - Easy to move between points in same cell.
  - easy to move to adjacent cells
  - Adjacency is easy to define
  - Cells are disjoint: $c_i \cap c_j = \emptyset, \quad W = \sum_i c_i$



Cells are labeled as

- Empty
- Occupied

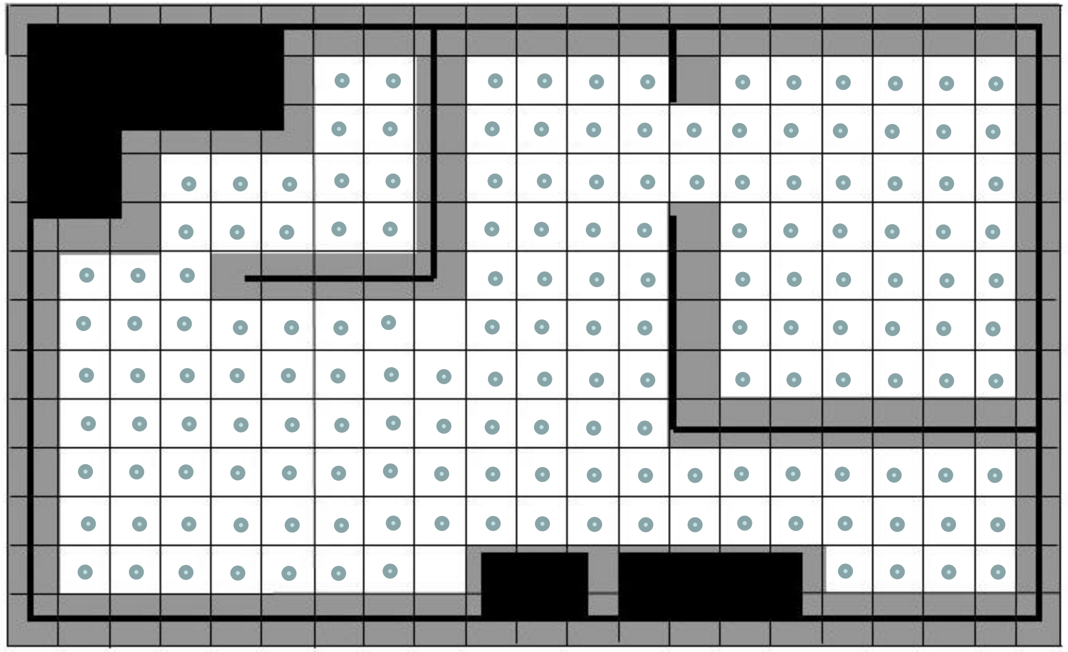
In known environment:

- Use geometric model to divide into cells & occupancy

In unknown environment:

- Use occupancy grid SLAM (e.g., “gmapping”)

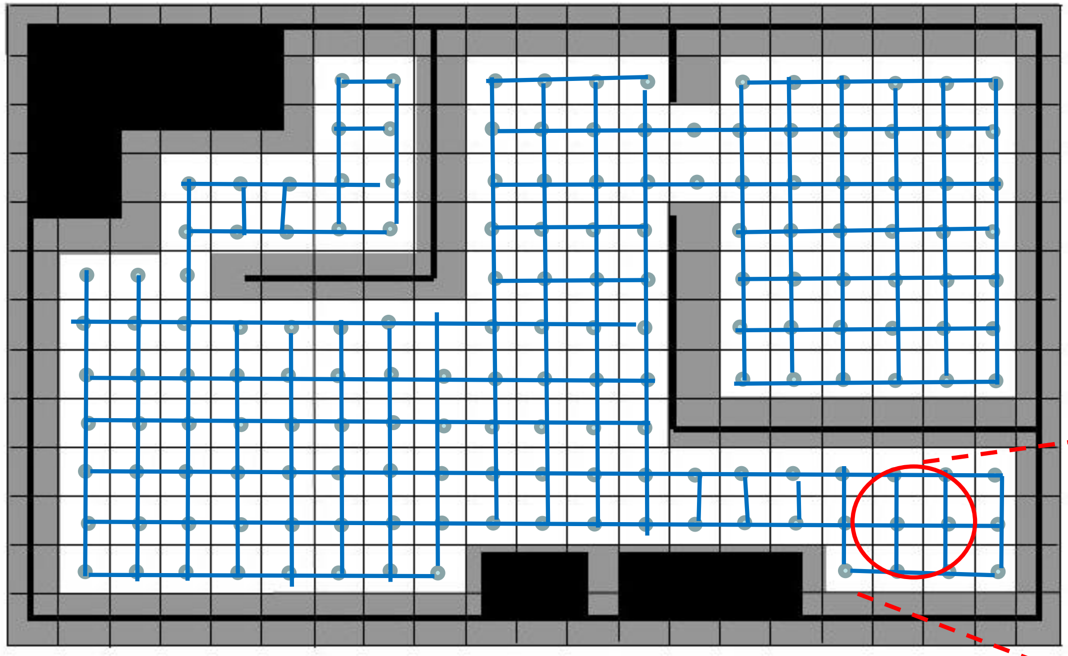
Application to Autonomous Planning



Adjacency Graph

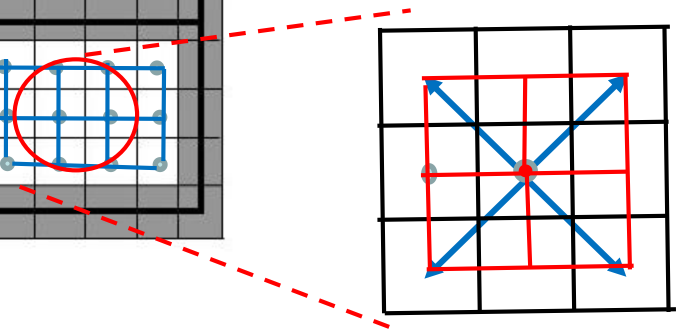
- Node: empty/free cells
- Edges: transitions between adjacent free cells

Application to Autonomous Planning



Adjacency Graph

- Node: empty/free cells
- Edges: transitions between adjacent free cells



Shortest Path problem

Minimize $\left(w_{i_1,j_1} + \cdots + w_{i_p,j_p}\right)$ such that $x_{start} \in c_{i_1,j_1}$, $x_{final} \in c_{i_p,j_p}$

Finding the Optimal Policy

Recursive Derivation: **Step 1**

- $T = 1$ (greedy solution): $\pi_1(x) = \underset{u}{\operatorname{argmax}}\, r(x,u)$
- The *value* (or *cost-to-go) function* describes the "value" of the cumulative reward when the optimal actions is taken:

$$V_1(x) = \max_u r(x,u) \quad (= \max_u E[r(x,u)],\ E \text{ dropped below})$$

Recursive Derivation: **Step 2**

- $T = 2$: $\pi_2(x) = \underset{u}{\operatorname{argmax}}[r(x,u) + \gamma \sum_z V_1(z) T(z|u,x\,)]$
- Value function at $T = 2$

$$V_2(x) = \max_u \left[r(x,u) + \gamma \sum_z V_1(z) T(z|u,x) \right]$$

Finding the Optimal Policy

Recursive Derivation: **Step T**

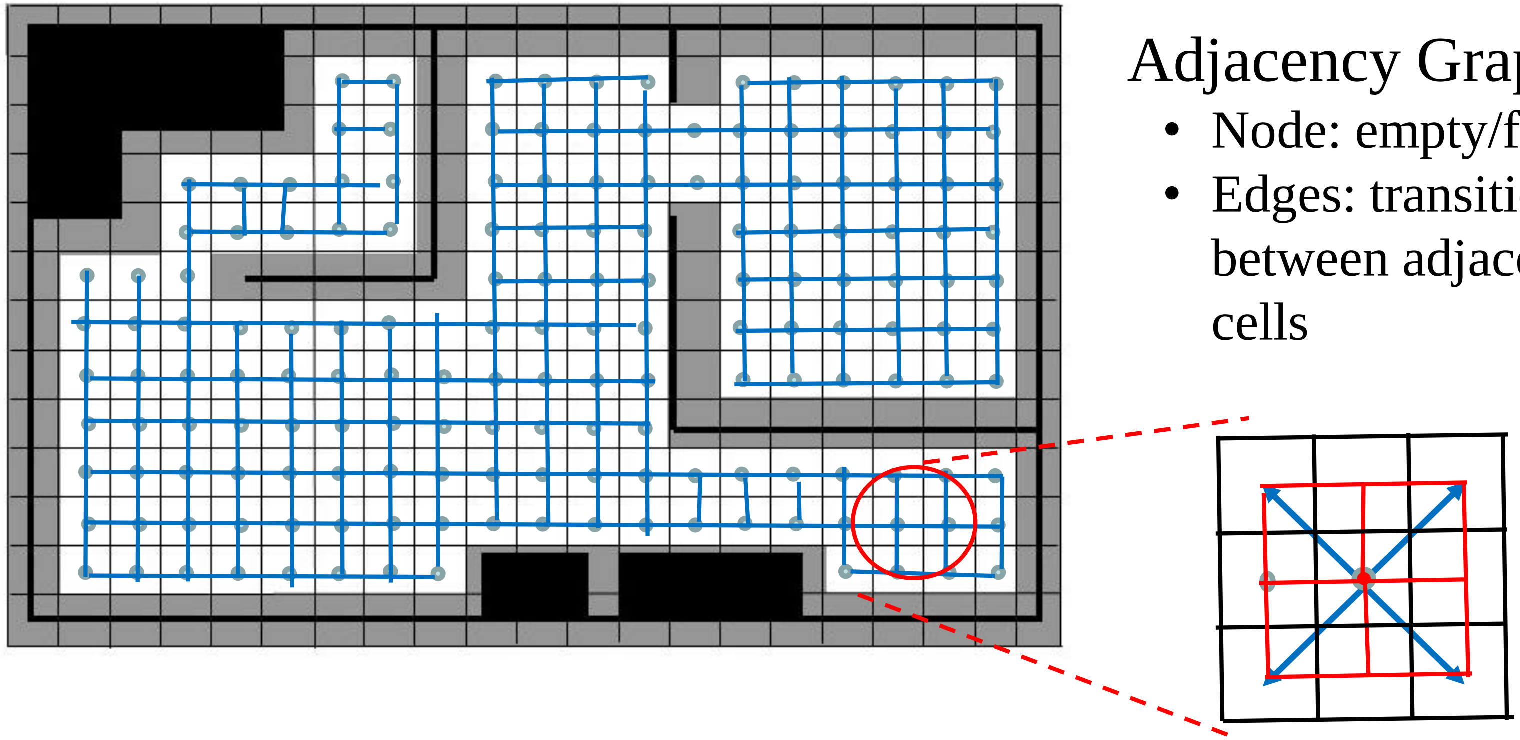
- $\pi_T(x) = \underset{u}{\text{argmax}}[r(x,u) + \gamma \sum_z V_{T-1}(z) T(z|u,x)]$
- $V_T(x) = \max_u [r(x,u) + \gamma \sum_z V_{T-1}(z) T(z|u,x)]$

Infinite Horizon:

- $V_\infty(x) = \max_u [r(x,u) + \gamma \sum_z V_\infty(z) T(z|u,x)]$
- The “Bellman Equation”
- The optimal value function is the “*fixed point*” of this equation. This is the basis of *“value iteration”*
- The optimal policy (at any time)

$\pi^*(x) = arg \max_u [r(x,u) + \gamma \sum_z V_\infty(z) T(z|u,x)]$=

Application to Autonomous Planning



Adjacency Graph

- Node: empty/free cells
- Edges: transitions between adjacent free cells

Shortest Path problem

Minimize $\left(w_{i_1,j_1} + \cdots + w_{i_p,j_p}\right)$ such that $x_{start} \in c_{i_1,j_1}$, $x_{final} \in c_{i_p,j_p}$

Graph Search: the A* algorithm

General Graph Search Goal: *search* the (adjacency) graph for a *feasible* path connecting the start to the goal node(s).

Optimal Search: find the feasible path with the guaranteed lowest cost of traversal (the sum of the edge weights along the path)

General Graph Search data structures:

- All states or nodes are labeled *unvisited, visited, dead*
- **Q:** a *priority queue*
- **T:** a *spanning tree* or *search tree*

General Graph Search Algorithm:

- **Init:** mark x_{init} *visited*, all other states visited
  insert x_{init} into **Q**
  insert x_{init} into **T**

Graph Search: basic algorithm structure

- **While Q not empty:**
  - x_i = ***getFirst*(Q)**
  - **If** $x_j = x_{goal}$,
    - Add pointer from x_j to x_i in **T**
    - Return **Success**
  - **For all** $u_j \in U(x_i)$ % get *successor* nodes
    - $x_j = \mathrm{f}(\mathrm{u_j})$
    - **If** x_j *not visited,*
      - mark x_j as *visited*
      - Add pointers from x_j to x_i in **T**
      - Insert x_j into **Q**
    - **Else** resolve duplicate links (if appropriate)
- Return **Failure**

Graph Search: A* algorithm

A* uses additional functions to improve its operation and outcome

- $g(x)$: *cost-to-arrive.*
  - The total edge cost from the start node to the current node x along an *optimal path*
- $h(x)$: *heuristic cost-to-go.*
  - An estimate of the cost between current node x and x_{goal}
- $k(x, x') =$ distance from node x to node x'
- $f(x) = g(x) + h(x)$: the estimated cost to the goal through x

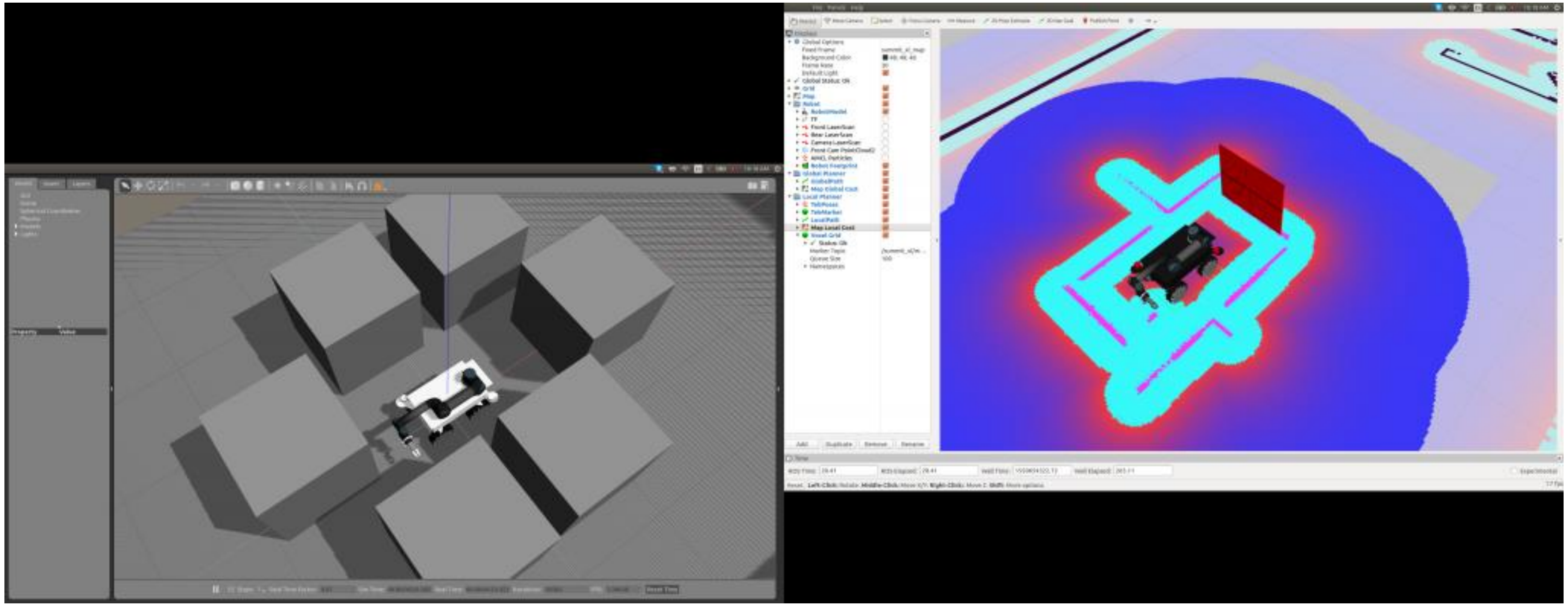
Summary of A*:

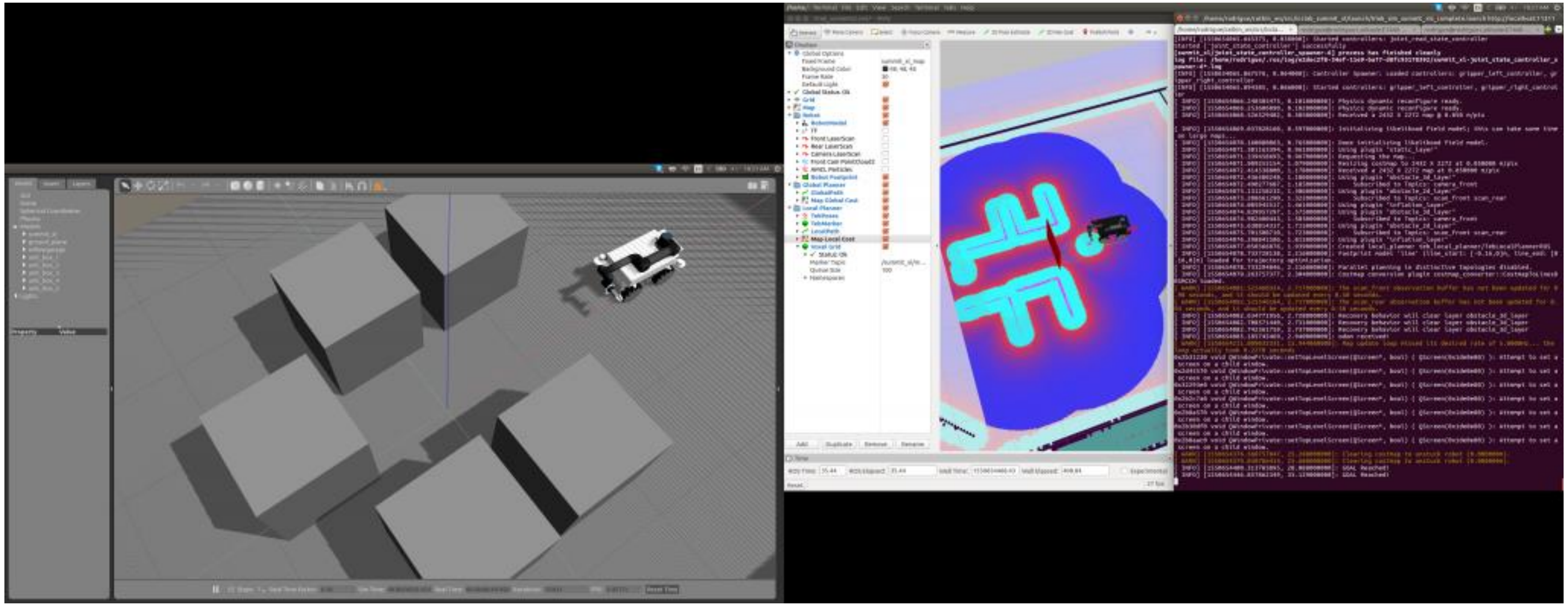
- ***getFirst***(**Q**) removes node x_k from **Q** with lowest $f(x_k)$
- For each successor node of x_k (denoted by x') removed from **Q**, check to see if going through x_k is a lower cost way to reach x'

Graph Search: A* algorithm

Replace the successor node processing loop with the following

- **For each** successor node of x_k (denoted by x')
  - $g_{test}(x') = g(x) + k(x, x')$; $f(x') = g(x') + h(x')$
  - **If** x' *visited,*
    - **If** $g_{test}(x') \leq g(x')$ % found a better path
      - Remove existing back-pointer from x' in **T**
      - Add back-pointer from x' to x_k in **T**
      - Add x' to **Q**
    - **Else** discard x' (or put x' on the CLOSED list)
  - **Else** **%** x' has not been visited
    - $g(x') = g_{test}(x')$
    - Add back-pointer from x' to x_k in **T**
    - Add x' to **Q**





ROS Goals for Next Week

GO through the steps 5, 6, 7, 8 of the *Core ROS Tutorial* at the beginner's level.

- You may prefer to the analogous steps in "*A Guided Journey to the Use of ROS*"

Download, install, *move_base*

Read about and Install Rviz

Heads-up: need to have visualization of your vehicle in Rviz by the following week.