

# Chapter 10

## Motion Planning

### Part 2

#### 10.2 Representation and Search for Global Motion Planning



# Outline

- 10.2 Representation and Search for Global Motion Planning
  - 10.2.1 Sequential Motion Planning
  - 10.2.2 Big Ideas in Optimization and Search
  - 10.2.3 Uniform Cost Sequential Planning Algorithms
  - 10.2.4 Weighted Sequential Planning
  - 10.2.5 Representation For Sequential Motion Planning
  - Summary

# Solution Techniques

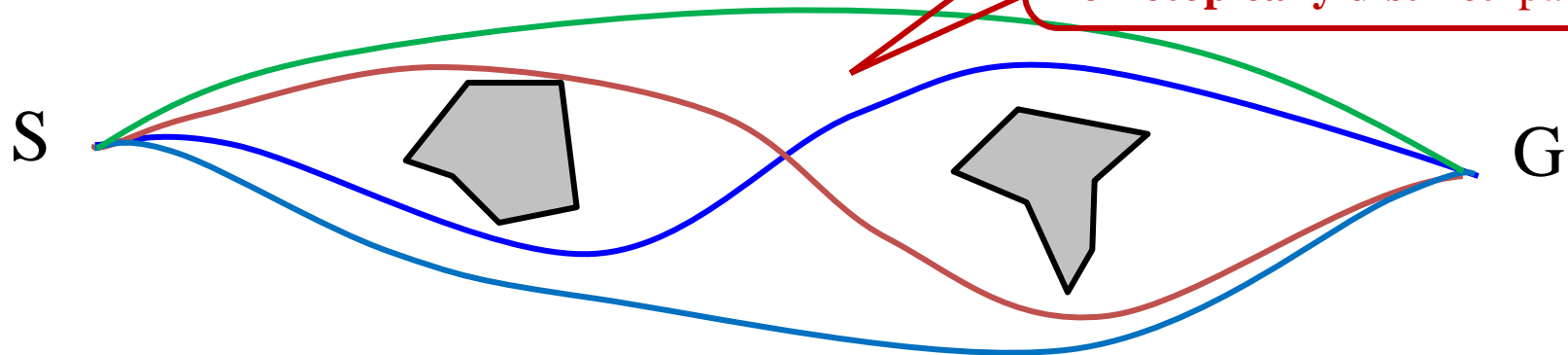
- Recall: Path planning is essentially an optimal control problem.
- Three clear solution techniques for optimal control:
  - Parameterization (how we did trajectory generation)
  - Variational (“geodesics”).
  - Dynamic Programming
- Why not just use trajectory generation algorithms?
  - We shall see....

# Outline

- 10.2 Representation and Search for Global Motion Planning
  - 10.2.1 Sequential Motion Planning
  - 10.2.2 Big Ideas in Optimization and Search
  - 10.2.3 Uniform Cost Sequential Planning Algorithms
  - 10.2.4 Weighted Sequential Planning
  - 10.2.5 Representation For Sequential Motion Planning
  - Summary

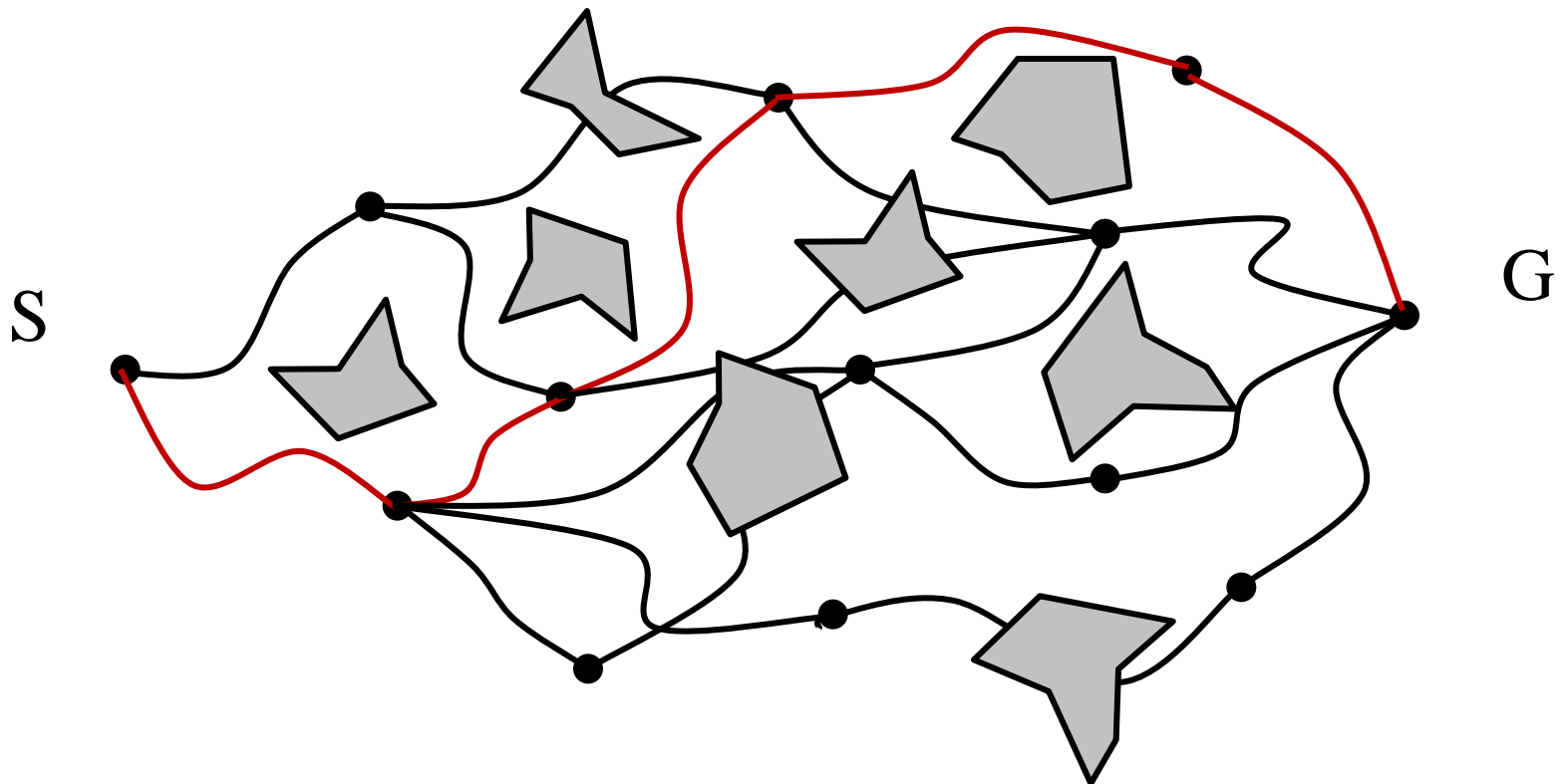
# 10.2.1.1 Why Not Continuum Methods?

- Too Many Solutions...
  - Scale is much larger. Many more solutions in some funny continuum sense.
- Too many Constraints...
  - Avoiding 1000 obstacles is 1000 constraints.
- Too many local minima.



# 10.2.1.2 Discretization of Search Spaces

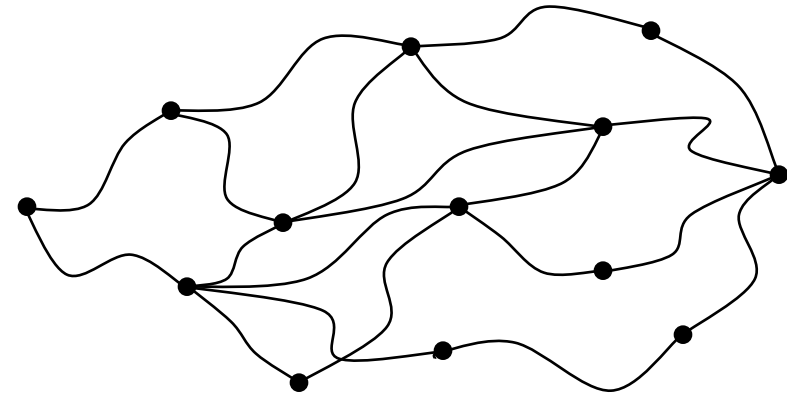
- **Embed a network** in space and search it (instead of space itself)..



# 10.2.1.2 Discretization of Search Spaces

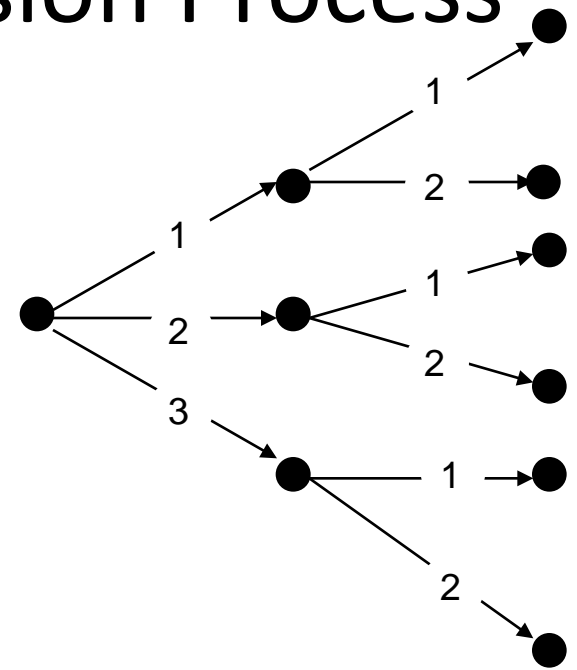
(State Discretization)

- Discrete states may or may not be regularly arranged.
- Join nearby states with edges.
- Produces a graph embedded in (i.e. a subset of) workspace or  $C$  space.
- Planning paths ...
  - in the continuum
- ... has become ...
  - reduced to a graph search problem.



# 10.2.1.3 Sequential Decision Process

- The **solution is a sequence** of small paths.
- Require, at each state encountered, **some options** for how to proceed.
- **Each option transitions** to a new state:



$$x_{k+1} = f(x_k, u)$$

Usually want these to be feasible

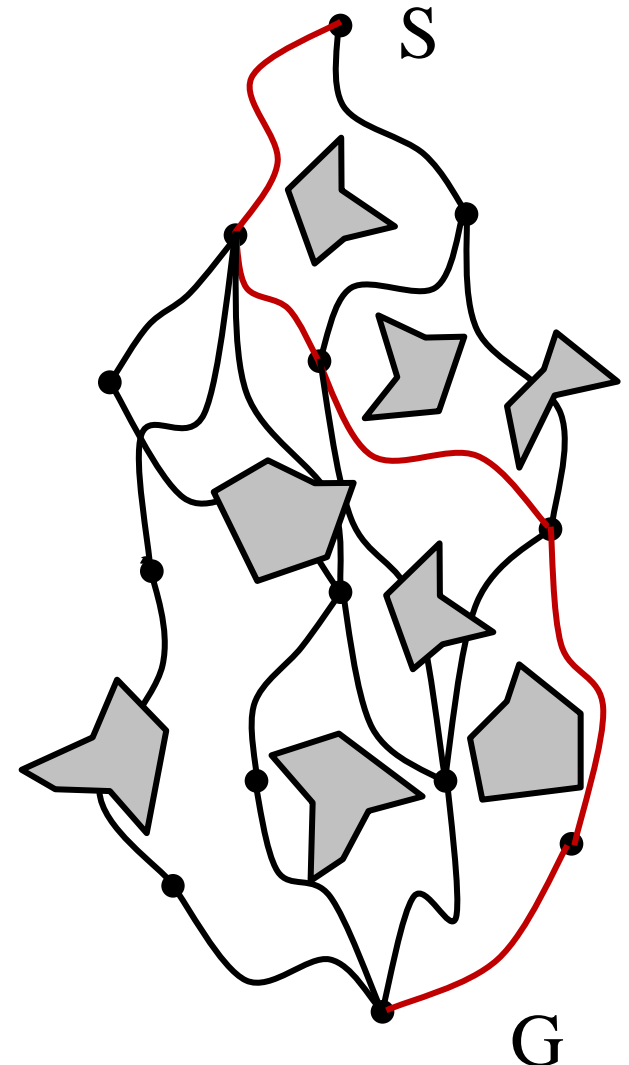
Maybe want them to avoid obstacles.

Sometimes, don't even check the edges for collision if they are short. Check only states,



# Discrete Motion Planning Formulation

- Given:
  - a graph
  - a start state
  - a goal state
- Find a **sequence of edges** (equivalently, states) connecting start to goal.
- Some formulations have multiple goals or goal regions.
- Some have multiple start states (uncertainty).

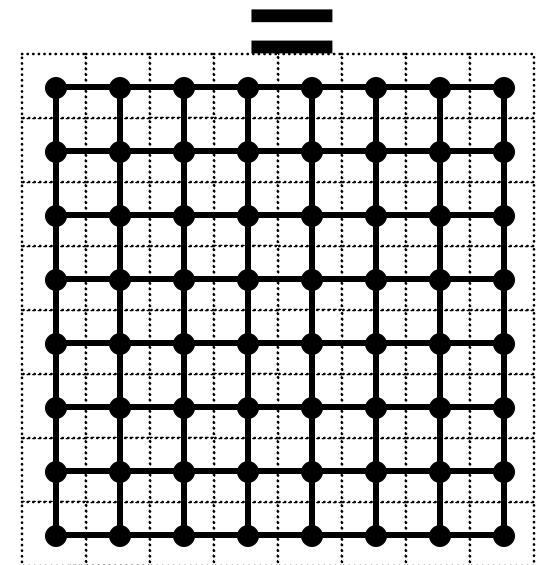
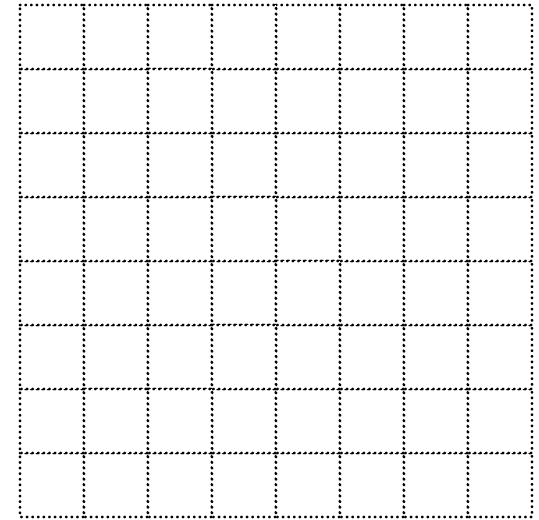


## 10.2.1.4 World Model, C-Space, and Search Graph (Discrete Representations)

- Convenient for performing sequential search.
- Abstract the continuum in two ways:
  - 1) Discretize the state space
  - 2) Discretize the motions so that they connect only the (nearby) states.
- Sometimes we do this based on knowledge of:
  - neither obstacle nor mobility (grid)
  - mobility ignoring obstacles (state lattice)
  - obstacles ignoring mobility (Voronoi diagram, roadmaps)

# 10.2.1.4 World Model, C-Space, and Search Graph (Grids & Lattices as Graphs)

- Search algorithms defined on **networks**.
- Grids and lattices are just regular arrangements of states.
- ANY algorithm defined on a network can be implemented on a **grid**.
- Edges may be implicit but they are always there.

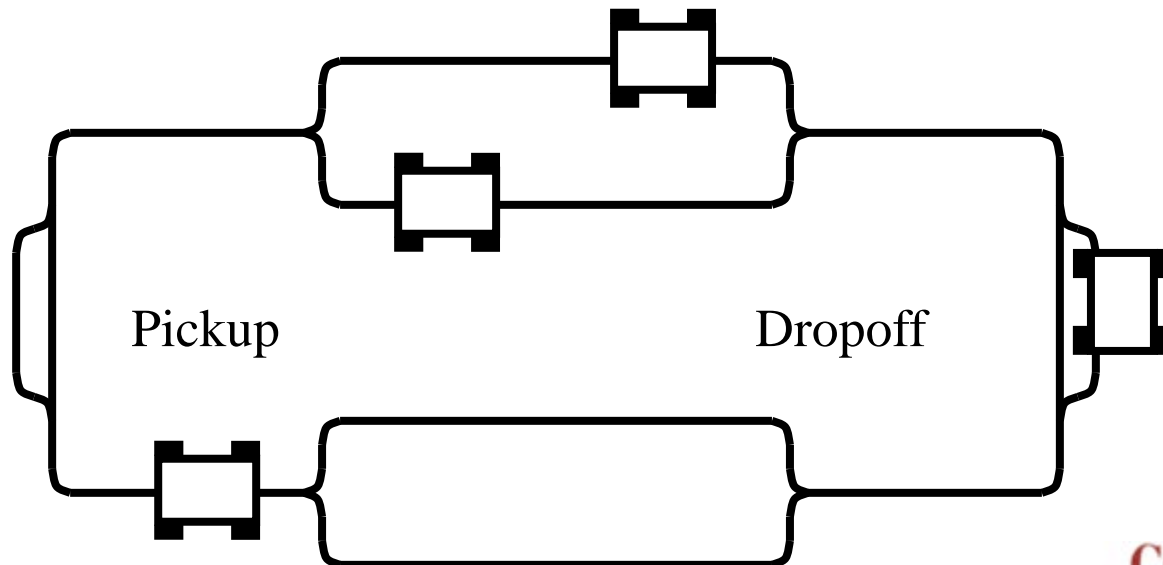


# 10.2.1.5 Search Space Design

- Implicit edges defer the motion generation problem post-planning.
  - Works sometimes. System must be predictable.
  - However, sometimes constraints must be represented to avoid failure.
- Tradeoff is **search convenience vs constraint convenience**.
- Discrete obstacles can be encoded in search space.
  - by **removing edges**.
  - Otherwise, need **cost field**.
- Often search space is **generated on the fly** but in rare cases, like a real road network, its known beforehand....

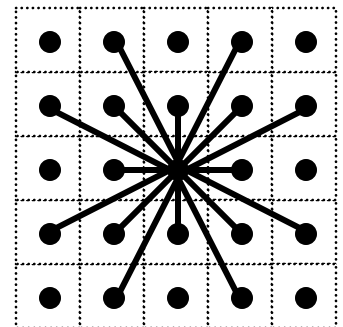
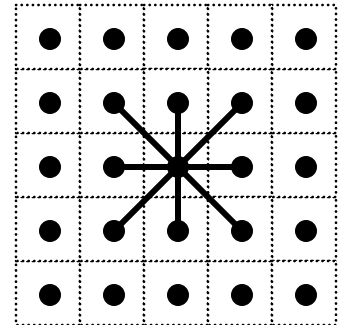
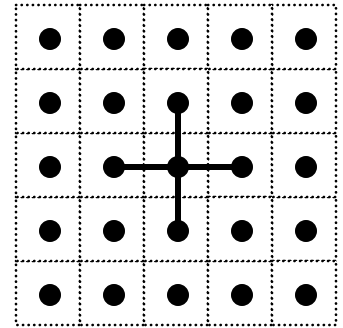
# 10.2.1.5.1 Road Networks

- Consider representing arbitrary **free paths**
  - perhaps related in some network (joining at intersections).
  - maybe not maximally distant from obstacles like Voronoi
- We **impose constraints of allowable motions first, and worry about obstacles second** (as was done in some forms of obstacle avoidance).



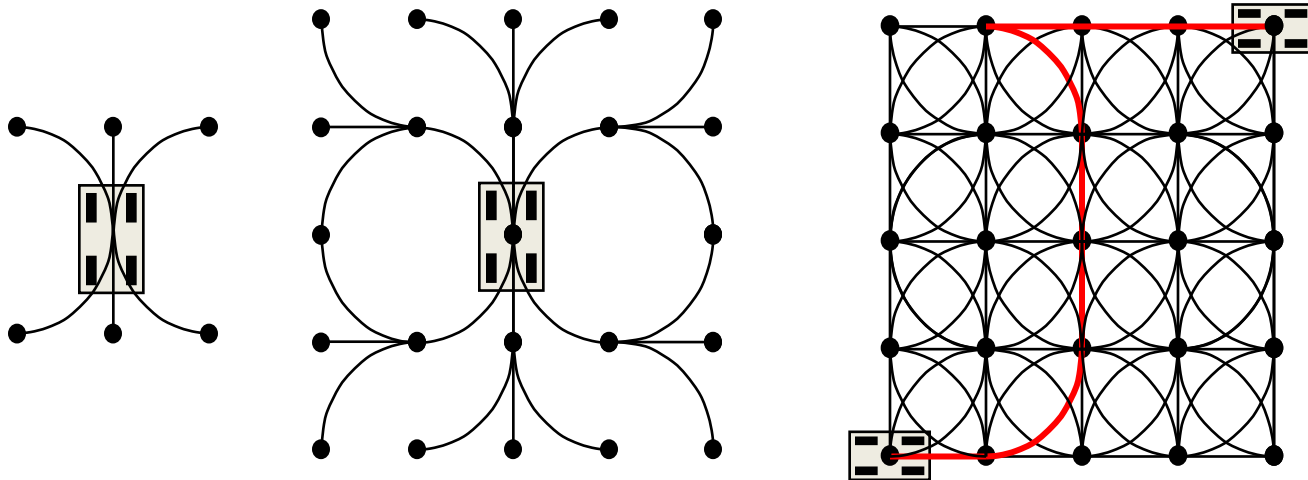
## 10.2.1.5.2 Workspace Lattices

- Search algorithms are defined on **networks**.
- **Grids and lattices** are just regular arrangements of states.
- ANY algorithm defined on a network can be implemented on a grid.
- Edges may be implicit but they are always there.



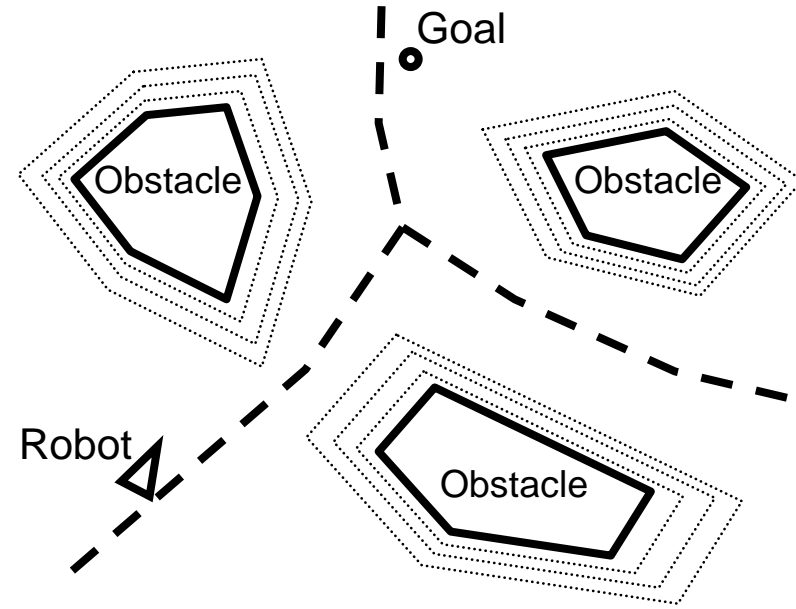
## 10.2.1.5.3 State Lattices

- Enforce differential constraints directly in the search space.
- For example, Reeds-Shepp car. Require heading continuity across nodes.



# 10.2.1.5.4 Voronoi Diagrams

- Set of all points which are equidistant from **at least** two obstacle boundaries.
- Local maxima in the proximity field.
- Can be generated from a field representation with the “distance transform”.





# Outline

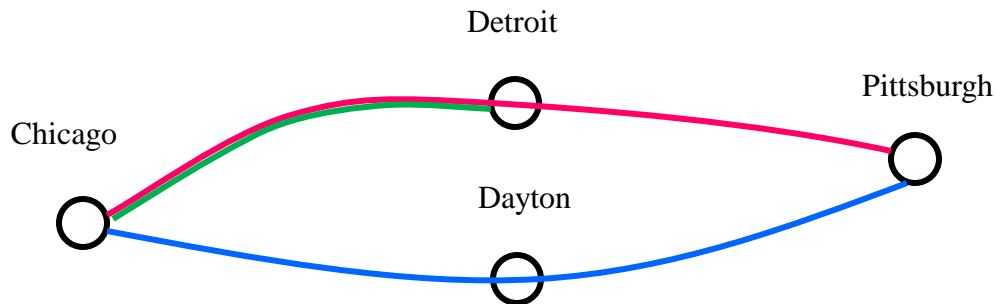
- 10.2 Representation and Search for Global Motion Planning
  - 10.2.1 Sequential Motion Planning
  - 10.2.2 Big Ideas in Optimization and Search
  - 10.2.3 Uniform Cost Sequential Planning Algorithms
  - 10.2.4 Weighted Sequential Planning
  - 10.2.5 Representation For Sequential Motion Planning
  - Summary

# 10.2.1.1 Principle of Optimality

(Bellman 60s)

- The basis of the famous and very useful Dynamic Programming Algorithm.
- Applies to sequential (aka Markov) decision processes (SDP).

**“An optimal policy has the property that whatever the initial state and initial decision are, the remaining decisions must constitute an optimal policy with regard to the state resulting from the first decision.”**



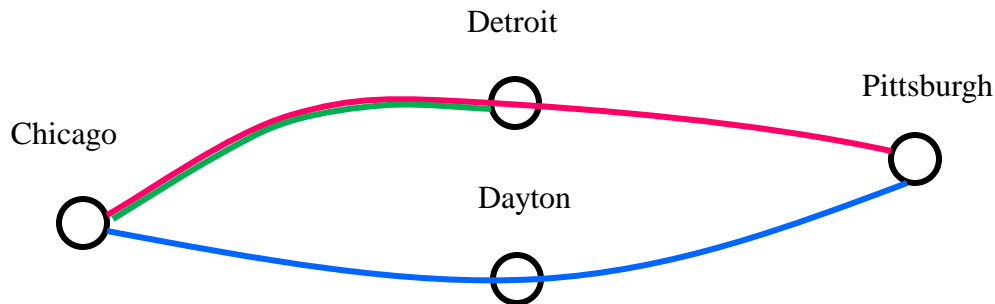
If the (whole) red path is optimal ...

the green one must also be optimal.

# 10.2.1.1 Principle of Optimality

(Bellman 60s)

- Dynamic Programming...
  - A large class of programming algorithms that are based on breaking a large problem down (if possible) into incremental steps so that, at any given stage, optimal solutions are known sub-problems.



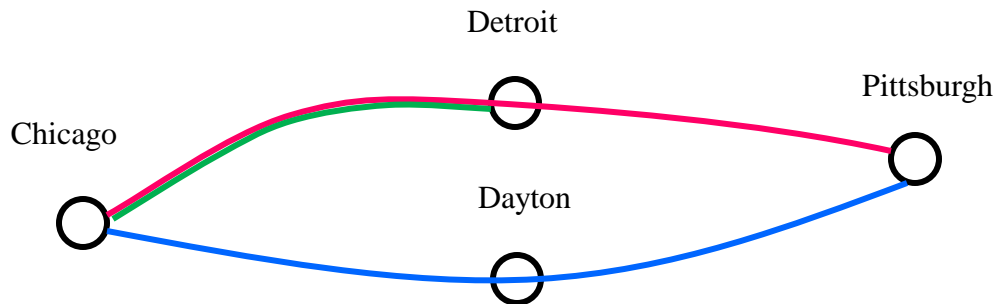
If the (whole) red path is optimal ...

the green one must also be optimal.

# 10.2.1.1 Principle of Optimality

(Notion of Proof)

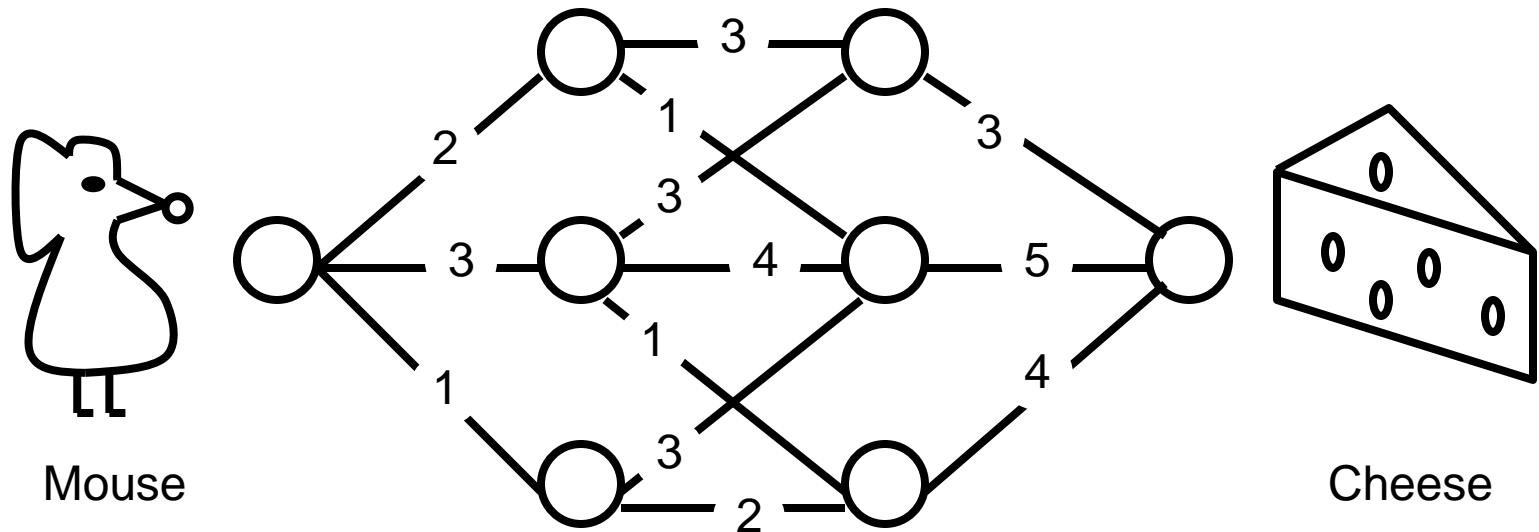
- Intuitively, the optimal solution to the entire problem must be composed of optimal solutions to the subproblems.
  - This only true for SDPs.
- Easy to prove by contradiction.....
  - Otherwise, you could substitute the optimal subproblem and generate a better solution.



# 10.2.1.1 Principle of Optimality

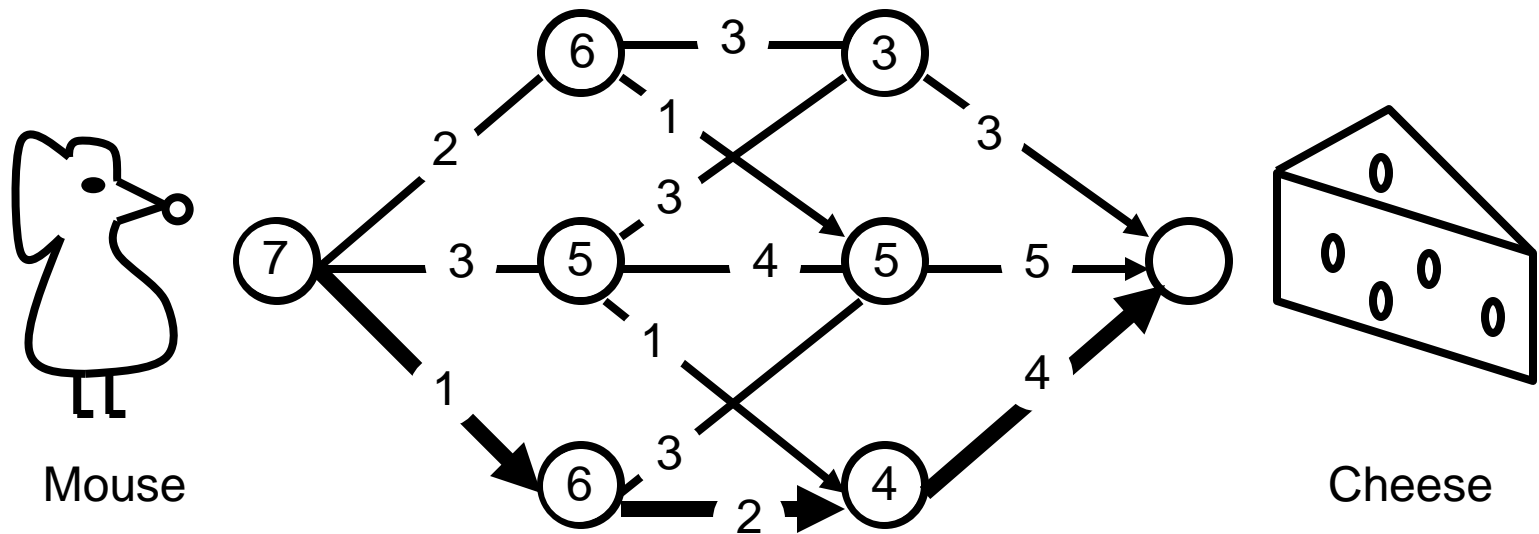
(Dynamic Programming)

- Starting at the start node, the mouse has to pick one of 3 and then one of (2 or 3) nodes...
- There are 7 possible paths of 3 edges (7 edges in middle phase).



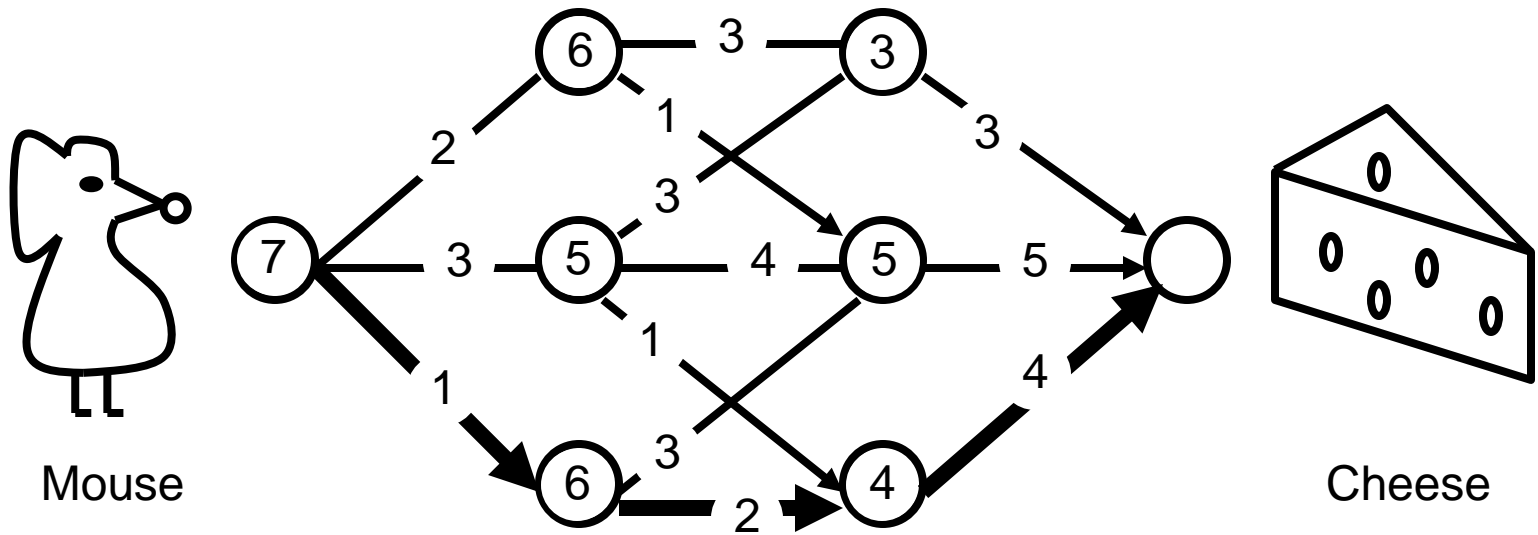
# 10.2.2.1.1 Backward Traversal

- To solve the problem, work backwards from the goal:
  - Label each node with the cost of the best path to the goal from there.
  - Record a “backpointer” to the next node in the forward direction.
  - Move backward one level at a time.



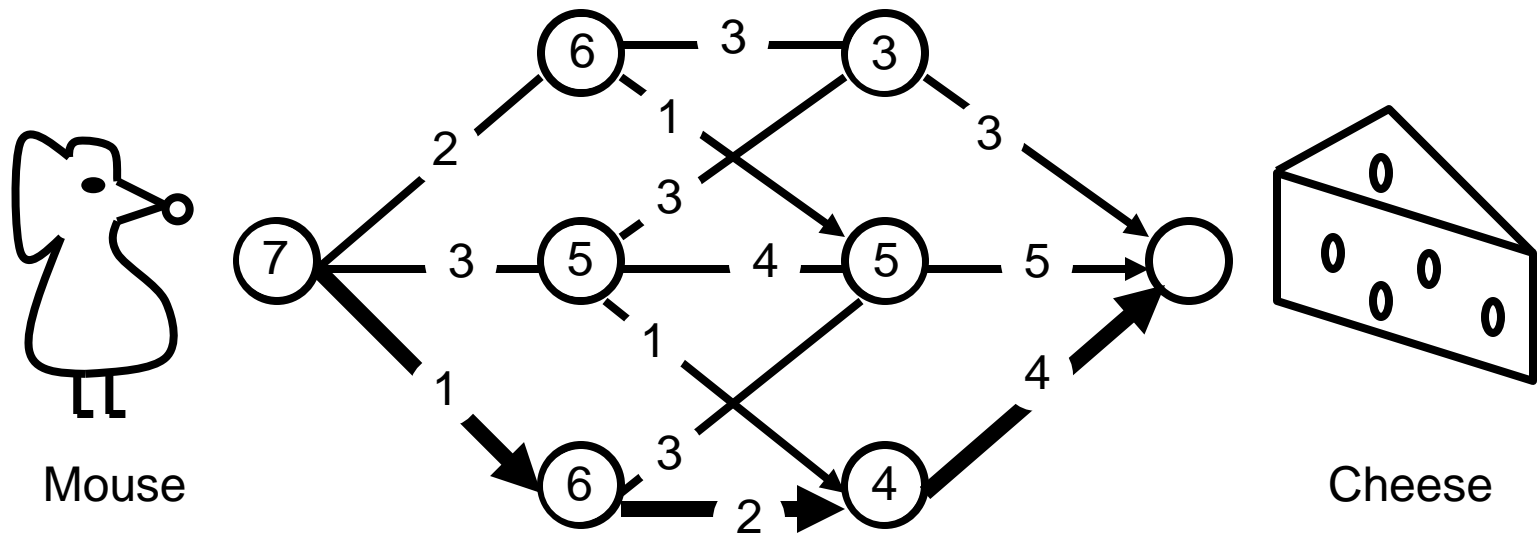
# 10.2.2.1.1 Backward Traversal

- Notice:
  - Brute force complexity is the number of distinct paths times the length of the paths (= 21 ops).
  - Dynamic programming complexity is the **number of edges** (=13).



# 10.2.2.1.1 Backward Traversal

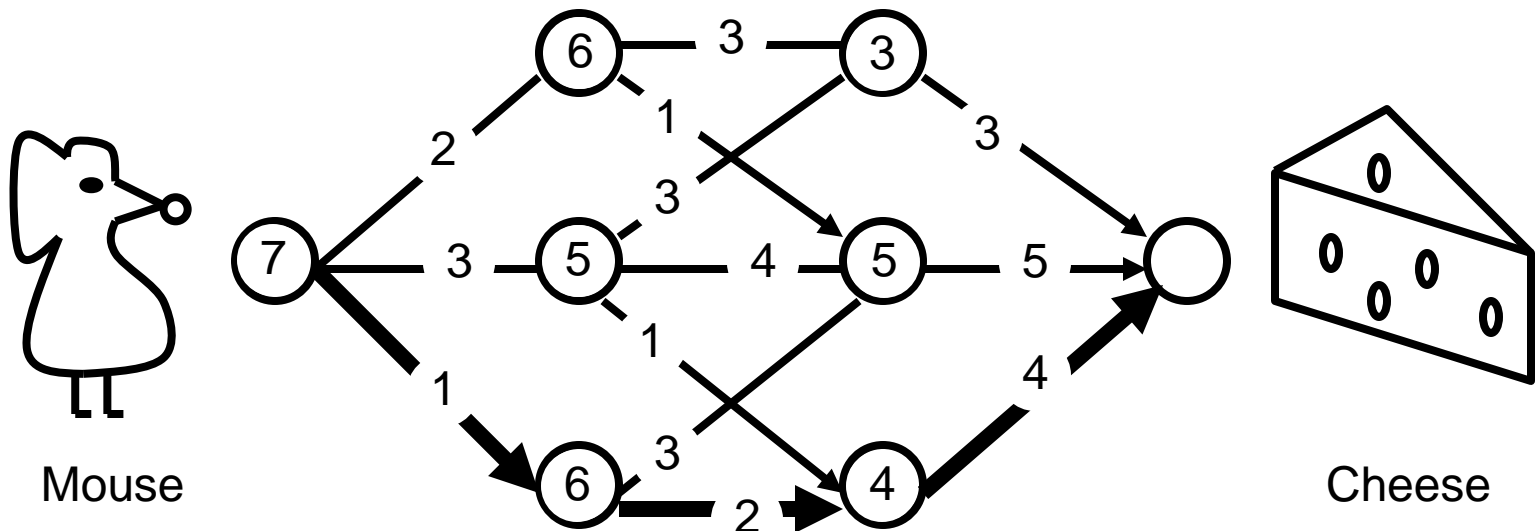
- Notice:
  - Decisions on backpointers are final → **commit as you go**.
  - **A spanning tree is constructed** in the process of graph traversal → each node can reach root on unique path.





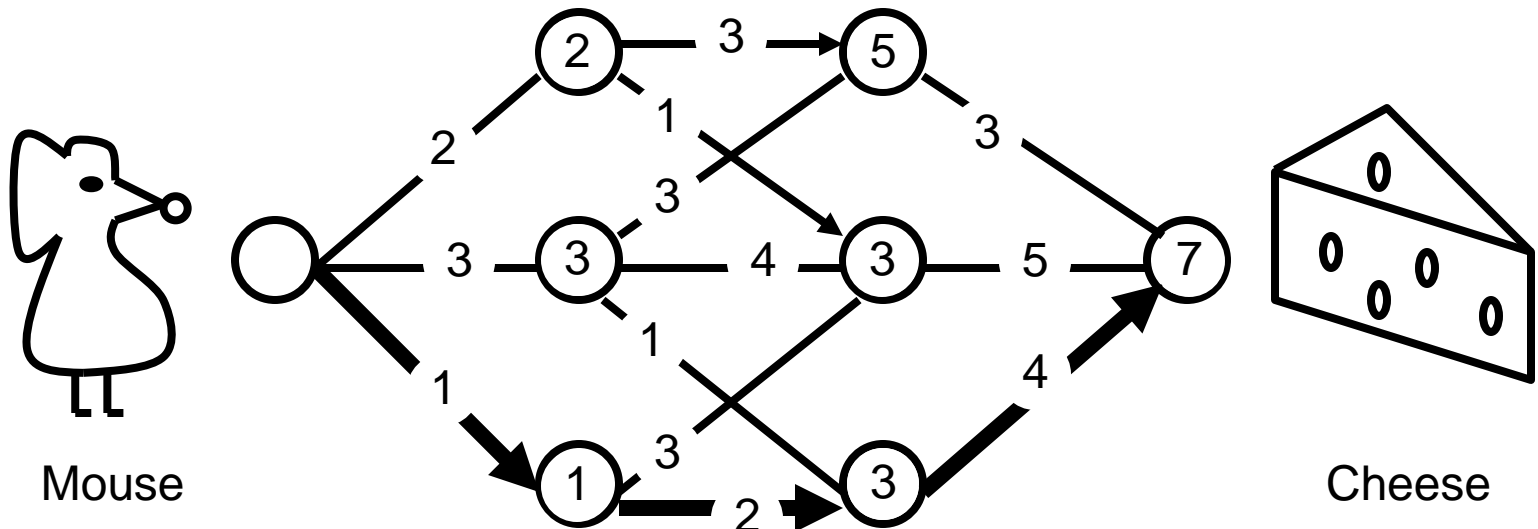
# 10.2.2.1.1 Backward Traversal

- Notice:
  - The nodes or states are a **convenient place to store** both ...
    - “best cost so far”
    - backpointers which record the sequential decisions.



# Forward Traversal

- Branching factor may make one direction preferable.
- That will not happen in locally connected graphs like those derived from grids.
- Here, “forward pointers” were remembered to make tree look the same as last example. Either option is OK for remembering the path.

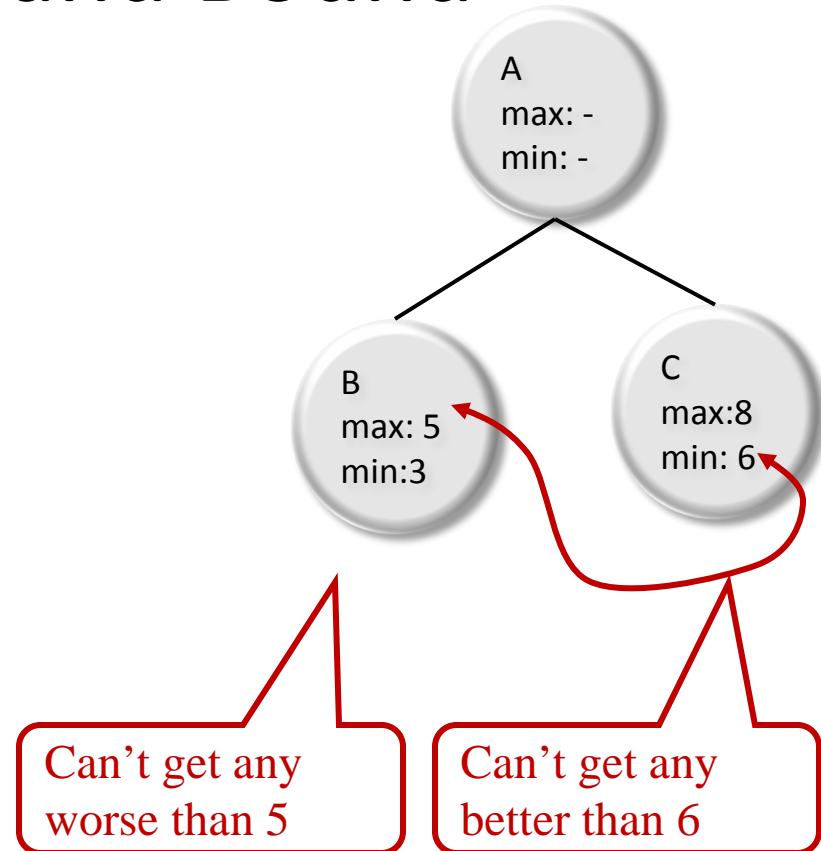


## 10.2.2.2 Branch and Bound

- Provides a way to **eliminate entire sections of the search space**.
- Relies on two ingredients:
  - A mechanism to **split up the search space** (branching)
  - A mechanism to **quickly compute bounds** on the quality of a solution at a node.

# 10.2.2.2 Branch and Bound

- Suppose:
  - looking for shortest path.
  - each node has a **max and a min bound on total path length** if they are used.
- **Node C opposite need never be expanded**
  - because B's worst case beats C's best case.

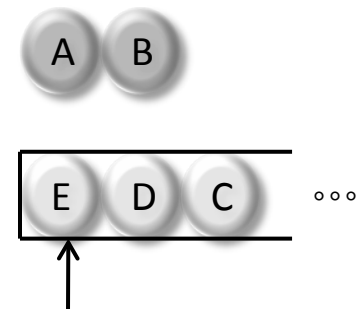
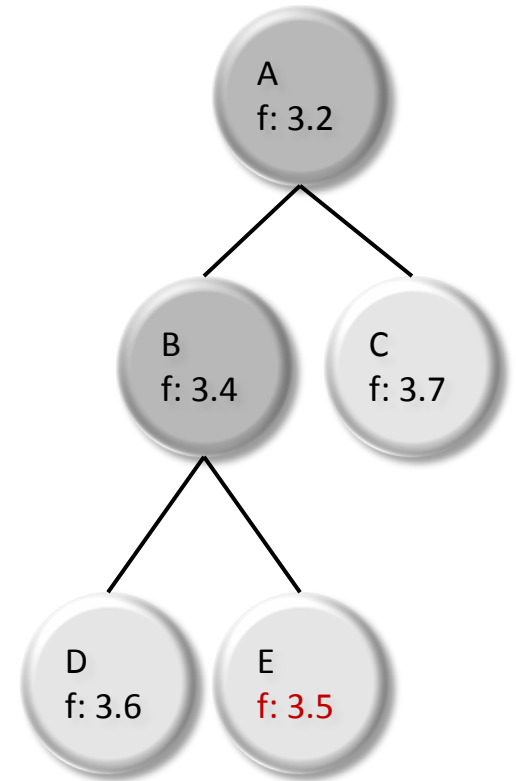


## 10.2.2.3 Best First Search

- Maintain all unexplored nodes in a **priority queue** and **expand the most promising node next**.
  - **Sort the queue** for fast ID of best
- Provides a way to encode arbitrary search strategies.
- Like Hill-climbing/steepest descent but:
  - Systematic – will eventually try all options.
  - May use smarter evaluation functions than local gradients.

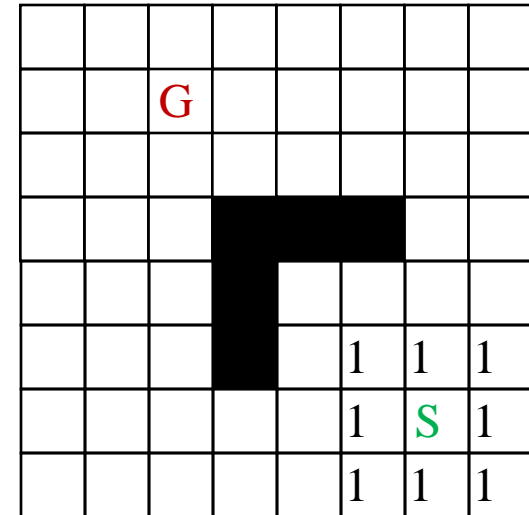
# 10.2.2.3 Best First Search

- Suppose:
  - looking for shortest path.
  - $F(\text{node})$  is an **estimate of the total path length** if the node is used.
- Darker nodes are closed.
- Expand node E next.



## 10.2.2.4 Policy Storage

- The path integral nature of path cost means **the optimal path to/from anywhere can be stored** in one compact structure.
- Store optimal potential or its gradient.



# 10.2.2.4 Policy Storage

		G						
					2	2	2	2
					2	1	1	1
					2	1	S	1
					2	1	1	1



# 10.2.2.4 Policy Storage

		G						
						3	3	
					2	2	2	2
					2	1	1	1
			3	2	1	S	1	
			3	2	1	1	1	

# 10.2.2.4 Policy Storage

		G					
					4	4	4
						3	3
					2	2	2
					2	1	1
		4			2	1	1
		4	3	2	1	S	1
		4	3	2	1	1	1

# 10.2.2.4 Policy Storage

		G		5	5	5	5
				5	4	4	4
						3	3
	5	5		2	2	2	2
	5	4		2	1	1	1
	5	4	3	2	1	S	1
	5	4	3	2	1	1	1

## 10.2.2.4 Policy Storage

- Goal is at distance 7 from start.
- Now know optimal path **from anywhere to the start.**
  - Or from start to anywhere.

9	8	7	6	6	6	6	6
8	8	7	6	5	5	5	5
7	7	7	6	5	4	4	4
6	6	6				3	3
6	5	5		2	2	2	2
6	5	4		2	1	1	1
6	5	4	3	2	1	S	1
6	5	4	3	2	1	1	1

# Outline

- 10.2 Representation and Search for Global Motion Planning
  - 10.2.1 Sequential Motion Planning
  - 10.2.2 Big Ideas in Optimization and Search
  - 10.2.3 Uniform Cost Sequential Planning Algorithms
  - 10.2.4 Weighted Sequential Planning
  - 10.2.5 Representation For Sequential Motion Planning
  - Summary

# Uniform Cost Edges

- Groundrules for the rest of this section...
- “Length” of the path is defined as the **number of edges** required to reach it.
  - Edges have **equal** length or cost.

# Reminder: Desiderata

- Complete:
  - Find a path if it exists
  - Report failure otherwise (i.e. terminate)
- Sound / Feasible:
  - Meet all constraints
- Optimal:
  - Produce optimal solution if any.

# 10.2.3.1 Wandering Motion Planner

- Virtues:
  - Stays on graph
  - Discovers graph
  - Fixed memory
- Problems:
  - Nonsystematic
    - May never terminate in practice
  - Certainly not optimal
    - May generate cyclic solutions
  - Does not remember the path for later execution.

```
00 algorithm wanderPlanner ()
01    $x \leftarrow x_s$ 
02   while (true)
03     for some( $u \in U(x)$ )
04        $x \leftarrow f(x, u)$ 
05       if( $x = x_g$ ) return success
06     endfor
07   endwhile
08   return
```

“Expand”  
the node

Aimless Wandering Planner

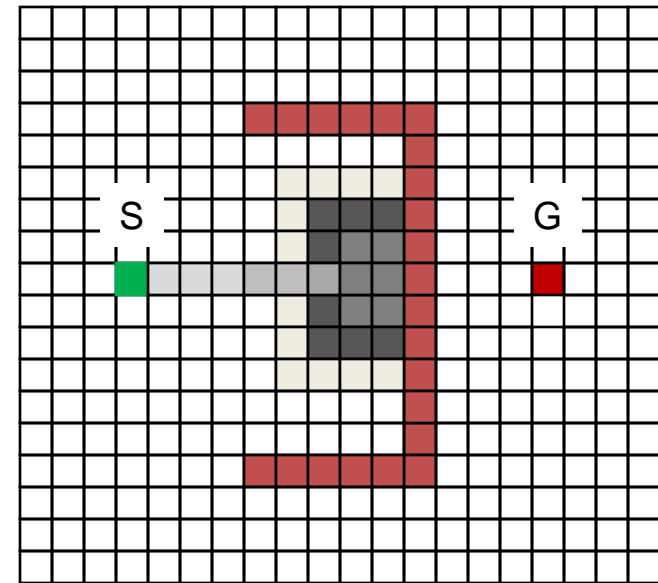
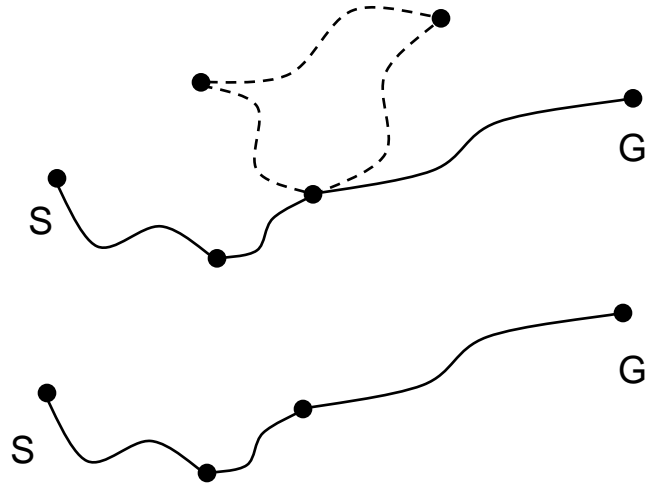
“Do random stuff (in simulation)  
until you stumble on the goal”



# 10.2.3.2 Systematic Motion Planner

(Busting Cycles)

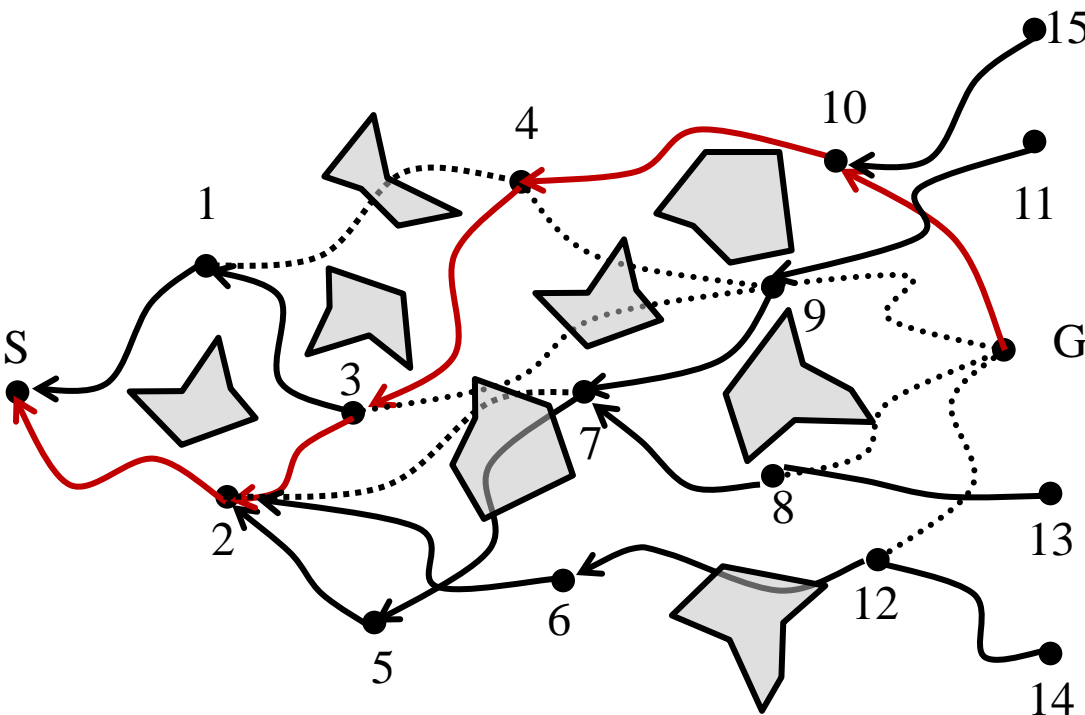
- Paths with cycles are better with cycles removed.
- Remember where you've been!!
- Side effect: filling up potential "wells".
- Systematic planners are **complete**.
- That takes **at least some memory**.
  - ... e.g. a horizon.



# 10.2.3.2 Systematic Motion Planner

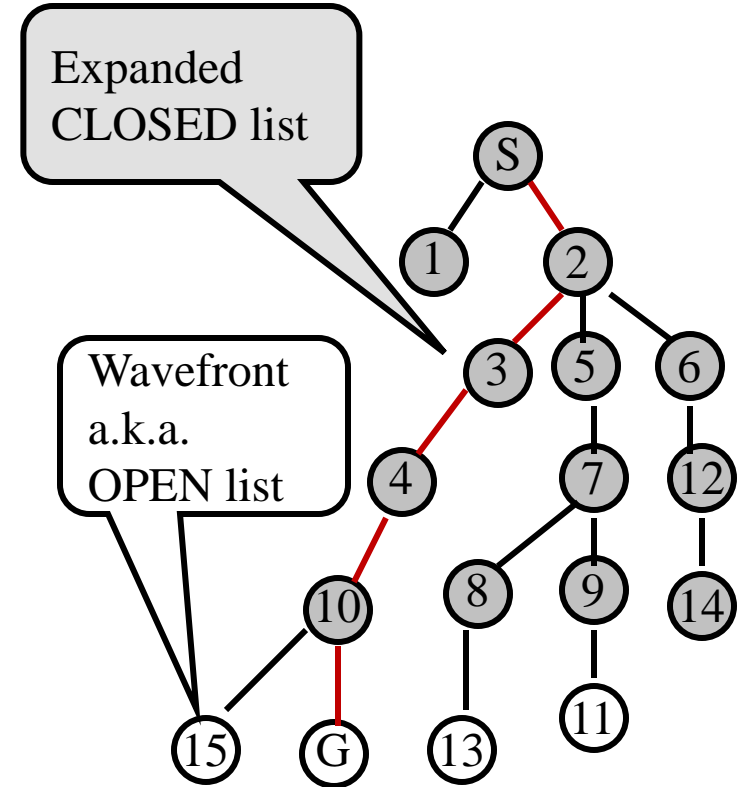
(Basics of Search)

- Build a spanning tree of the graph until you hit the goal.



**SEARCH GRAPH**

Usually **elaborated on the fly** to save memory

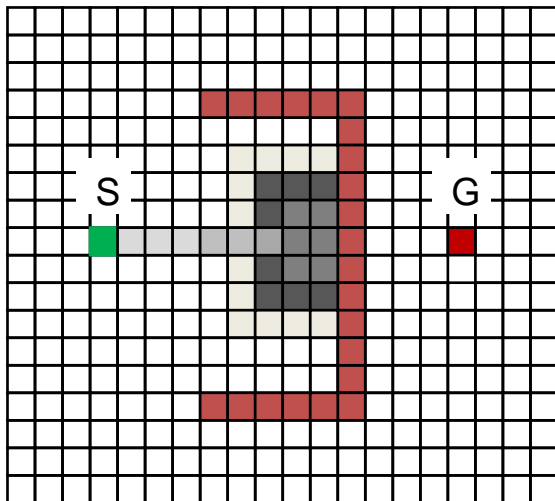


**SEARCH TREE**

Usually **encoded in “backpointers”**

# 10.2.3.2 Systematic Motion Planner

- Remembers all visited states.
  - Set O is the “open” (active) frontier.
  - Set C is the “closed” (inactive) set.



```
00 algorithm spanningTree( $x_s, x_g$ )
01  $O$ .insert( $x_s$ )
02  $x_s$ .parent ← null
03 while ( $O \neq \emptyset$ )
04      $x \leftarrow O$ .remove()
05      $C$ .insert( $x$ )
06     if (expandNodeST( $x$ )) return success
07 endwhile
08 return failure
```

Terminate on failure

Move state behind frontier

# Systematic Planner

- Remember parent pointers to enable path extraction.
- **Unique** parents creates spanning tree.
  - = acyclic cover
- Now need memory for O and C.
- Complete.
- However, **not optimal**.
  - Unless you **sort** the O set.

```
00 algorithm spanningTree ( $x_s, x_g$ )
01  $O$  .insert ( $x_s$ )
02  $x_s$  .parent  $\leftarrow$  null
03 while ( $O \neq \emptyset$ )
04      $x \leftarrow O$  .remove ( )
05      $C$  .insert ( $x$ )
06     if (expandNodeST( $x$ ) ) return success
07 endwhile
08 return failure
```

Markers in each state are an efficient alternative to using the set C

# 10.2.3.1 Systematic Motion Planner

(Node Expansion)

- Node expansion algorithm is similar in all of the algorithms here.

```
00  algorithm expandNodeST(x )
01  for each(u ∈ U(x ) )
02      xnext ← f(x, u)
03      if(xnext = xg )
04          xnext . parent ← x ; return success
05      else if(xnext ∉ O && xnext ∉ C )
06          xnext . parent ← x
07          O . insert (xnext )
08      endif
09  endfor
10  return failure
```

Solution !

Record  
Backpointer

State is new. Put  
it on frontier

# 10.2.3.3 Optimal Motion Planner

(BFS / Grassfire)

- **Sorted O set.** Code looks **identical** but...
- Set O becomes (ordered) FIFO queue.
- Removed at the front.
- Inserted at the back.

```
00 algorithm expandNodeBF(x)
01 for each(u ∈ U(x))
02   xnext ← f(x, u)
03   if(xnext = xg)
04     xnext.parent ← x; return success
05   else if(xnext ∉ O && xnext ∉ C)
06     xnext.parent ← x
07     O.insertLast(xnext)
08   endif
09 endfor
10 return failure
```

Insert at  
Back

# 10.2.3.3 Optimal Motion Planner

(BFS / Grassfire)

- Called **breadth first** search on graphs.
- Called **grassfire** on grids.
- Generally, the cost of optimality is the sorting.
  - But sorting is trivial (FIFO) **when edges are uniform cost.**

```
00 algorithm breadthFirst ( $x_s, x_g$ )
01  $O$  .insertLast ( $x_s$ )
02  $x_s$  .parent  $\leftarrow$  null
03 while ( $O \neq \emptyset$ )
04      $x \leftarrow O$  .removeFirst ()
05      $C$  .insert ( $x$ )
06     if (expandNodeBFx) return success
07 endwhile
08 return failure
```

Insert at  
back

Remove at  
front

“Wavefront optimality principle” =  
expand closest node to start.

# 10.2.3.3 Optimal Motion Planner

(BFS / Grassfire)

- Generate all paths of length one edge.
- Then all paths of length two edges...
- Each node is encountered first on the path which has least edges.

```
00 algorithm breadthFirst( $x_s, x_g$ )
01  $O$  .insertLast( $x_s$ )
02  $x_s$  .parent  $\leftarrow$  null
03 while ( $O \neq \emptyset$ )
04      $x \leftarrow O$  .removeFirst()
05      $C$  .insert( $x$ )
06     if (expandNodeBFx) return success
07 endwhile
08 return failure
```

Insert at  
back

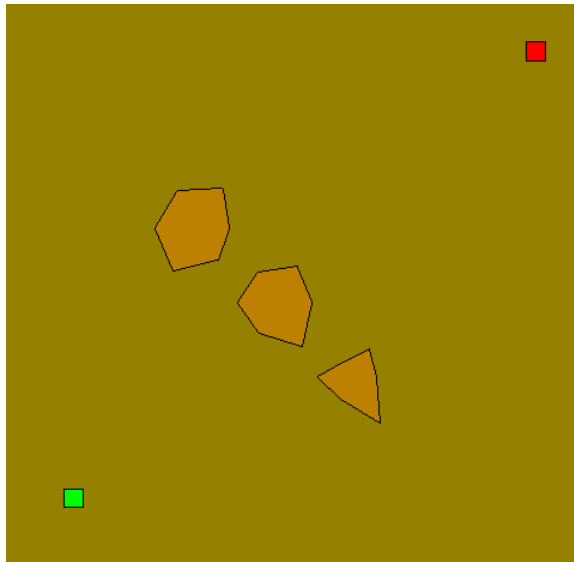
Remove at  
front

“Wavefront optimality principle” =  
expand closest node to start.

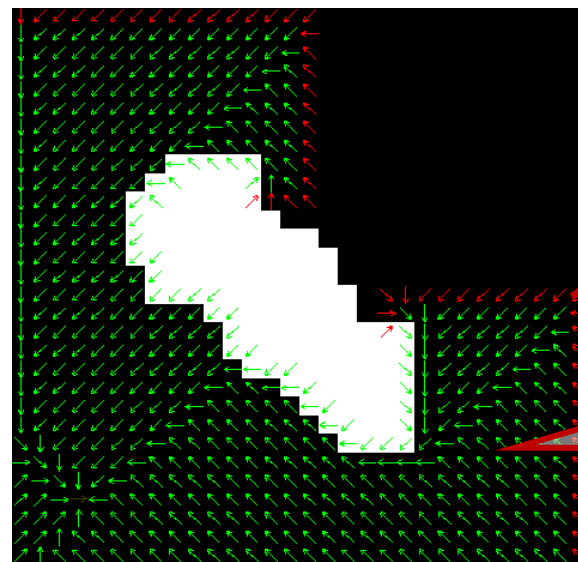


# 10.2.3.3 Optimal Motion Planner

(BFS / Grassfire)



World



Wavefront  
a.k.a.  
OPEN list

Expanded  
CLOSED list

Search Graph  
(backpointers)

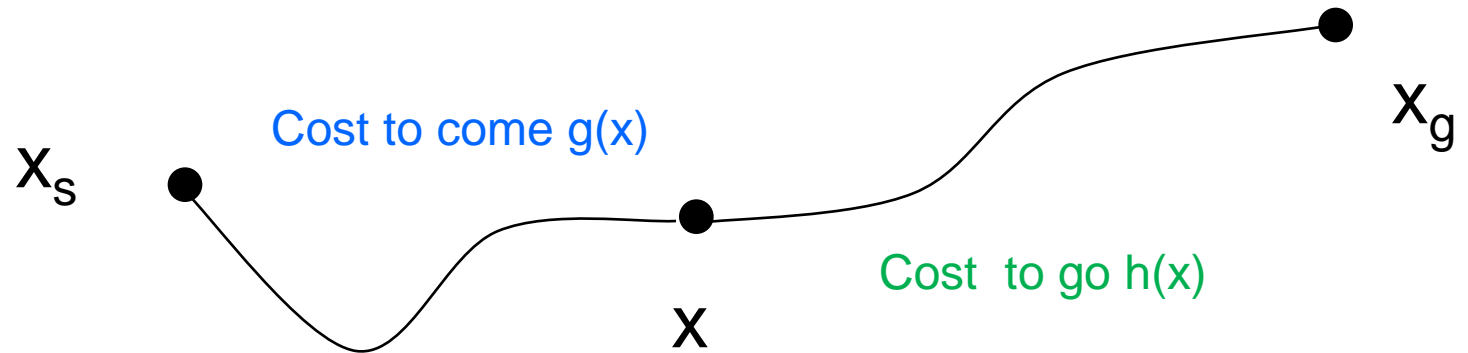
# Outline

- 10.2 Representation and Search for Global Motion Planning
  - 10.2.1 Sequential Motion Planning
  - 10.2.2 Big Ideas in Optimization and Search
  - 10.2.3 Uniform Cost Sequential Planning Algorithms
  - 10.2.4 Weighted Sequential Planning
  - 10.2.5 Representation For Sequential Motion Planning
  - Summary

# Nonuniform Cost Edges

- So far
  - ....“length” of the path was defined as the number of edges required to reach it.
- More generally let each edge have a variable, nonnegative cost.

# 10.2.4 Weighted Sequential Planning (Definitions)



# 10.2.4.1 Optimal Weighted Sequential Planner

(Dijkstra's Algorithm)

- $O$  becomes a **priority queue** sorted based on costs-to-come.
- Cost of states **expanded** increases monotonically
  - Added states must exceed cost of parent.
  - Queue is sorted.
- Invoke wavefront optimality principle.

```
00  algorithm Dijkstra ( $x_s, x_g$ )
01   $x_s.g \leftarrow 0$ 
02   $O.insertSorted(x_s, x_s.g)$ 
03   $x_s.parent \leftarrow null$ 
04  while ( $O \neq \emptyset$ )
05      $x \leftarrow O.removeFirst()$ 
06      $C.insert(x)$ 
07     if ( $x = x_g$ ) return success
08     expandNodeDijkstra( $x$ )
09  endwhile
10  return failure
```

# 10.2.4.1 Optimal Weighted Sequential Planner (Dijkstra's Algorithm)

- New issues:
  - Cost of nodes added is no longer monotone.
  - Therefore, costs of nodes added may not be optimal.
- So.....

```
00 algorithm calcG(x, xparent)
01   g ← xparent.g + edgeCost(x, xparent)
02   return g
```

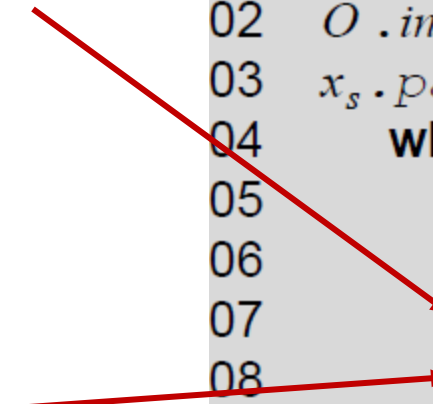
```
00 algorithm expandNodeDijkstra(x)
01   for each(u ∈ U(x))
02     xnext ← f(x, u)
03     gnew = calcG(xnext, x)
04     if(xnext ∉ O && xnext ∉ C)
05       addToODijkstra(xnext, x, gnew)
06     else if(gnew < x.g)
07       O.remove(xnext)
08       addToODijkstra(xnext, x, gnew)
09   endif
10   endfor
11   return
```

x<sub>next</sub>.g

# 10.2.4.1 Optimal Weighted Sequential Planner (Dijkstra's Algorithm)

- 1: Do actual sorting.
- 2: Delay\* test for goal to make sure its optimal.
- 2a: It used to be inside here for UCSP.

```
00  algorithm Dijkstra (xs, xg)
01  xs.g ← 0
02  O.insertSorted(xs, xs.g)
03  xs.parent ← null
04  while (O ≠ ∅)
05      x ← O.removeFirst()
06      C.insert(x)
07      if(x = xg) return success
08      expandNodeDijkstra(x)
09  endwhile
10  return failure
```



\* Its earlier in the code but it occurs later in time.

# 10.2.4.1 Optimal Weighted Sequential Planner (Dijkstra's Algorithm)

- 3: permit revisiting nodes
  - Update costs
  - Redirect parent pointers as necessary.
  - Remove and reinsert to keep  $O$  sorted.

```
00 algorithm expandNodeDijkstra( $x$ )
01 for each( $u \in U(x)$ )
02    $x_{next} \leftarrow f(x, u)$ 
03    $g_{new} = calcG(x_{next}, x)$ 
04   if( $x_{next} \notin O \ \&\& \ x_{next} \notin C$ )
05     addToODijkstra( $x_{next}, x, g_{new}$ )
06   else if ( $g_{new} < x.g$ )
07      $O.remove(x_{next})$ 
08     addToODijkstra( $x_{next}, x, g_{new}$ )
09   endif
10 endfor
11 return
```

$x_{next}.g$

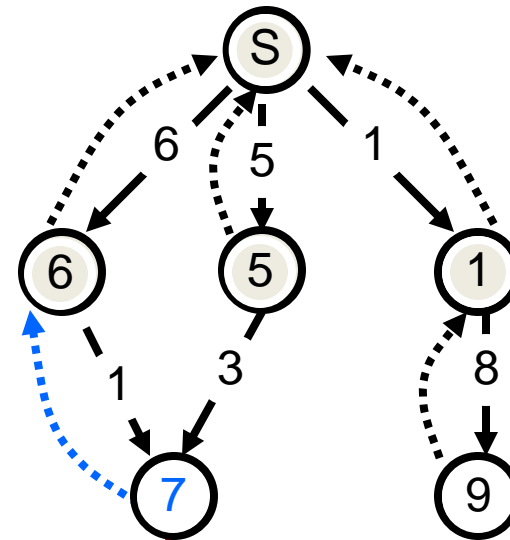
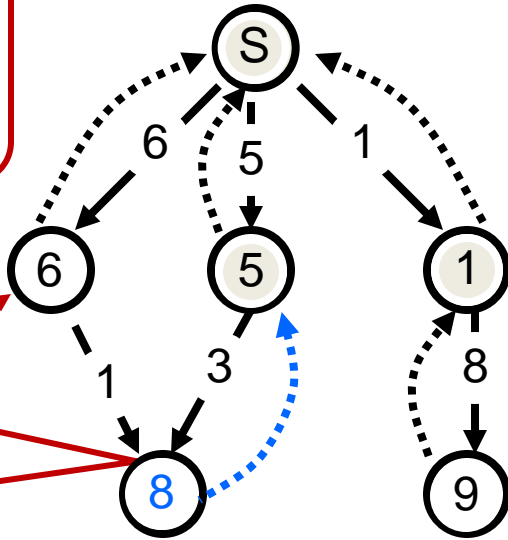
```
00 algorithm addToODijkstra( $x, x_{parent}, g$ )
01  $x.g \leftarrow g$ 
02  $O.insertSorted(x, g)$ 
03  $x.parent \leftarrow x_{parent}$ 
04 return
```



# 10.2.4.1 Optimal Weighted Sequential Planner (Backpointer Redirection on Open List)

1: In these diagrams, drawn nodes contain  $g()$  values.

2: Expansion of node  $g=6$  finds better path to node  $g=8$



—————> Original Edge  
 .....> Back Pointer

(n) Open Node  
 (n) Closed Node

3: Update  $g()$  and redirect backpointer

# 10.2.4.1 Optimal Weighted Sequential Planner (Dijkstra's Algorithm)

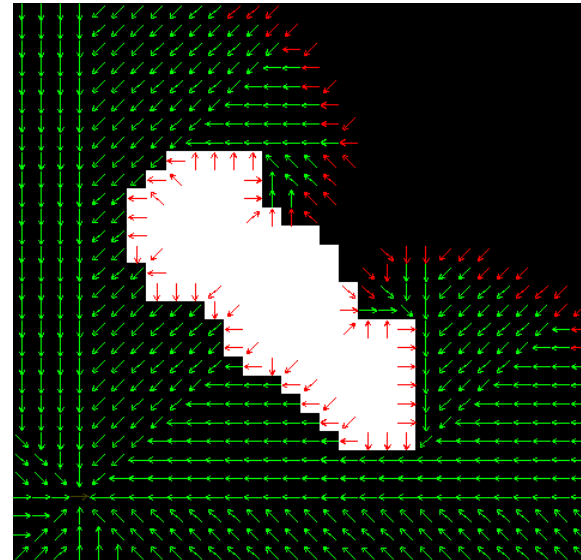
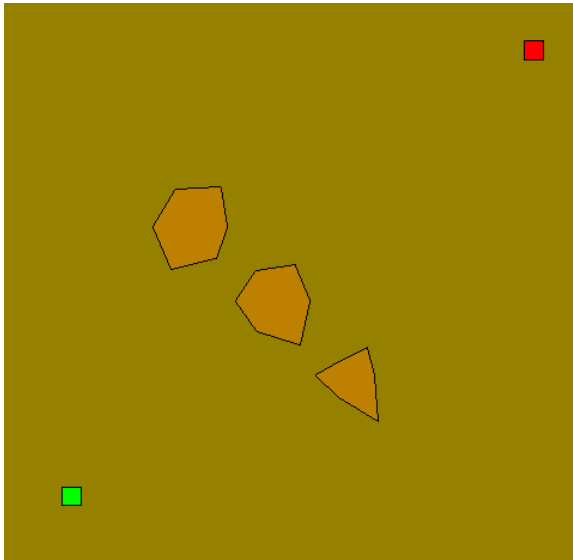
- Summary...
- Preserving optimality under nonuniform costs requires 3 things:
  - 1: Really sorting the queue.
  - 2: Delaying test for success.
  - 3: Tolerating and managing revisited nodes.

```
00 algorithm expandNodeDijkstra(x)
01 for each(u ∈ U(x))
02   xnext ← f(x, u)
03   gnew = calcG(xnext, x)
04   if(xnext ∉ O && xnext ∉ C)
05     addToODijkstra(xnext, x, gnew)
06   else if (gnew < x.g)
07     O.remove(xnext)
08     addToODijkstra(xnext, x, gnew)
09   endif
10 endfor
11 return
```

*x<sub>next.g</sub>*

```
00 algorithm addToODijkstra(x, xparent, g)
01 x.g ← g
02 O.insertSorted(x, g)
03 x.parent ← xparent
04 return
```

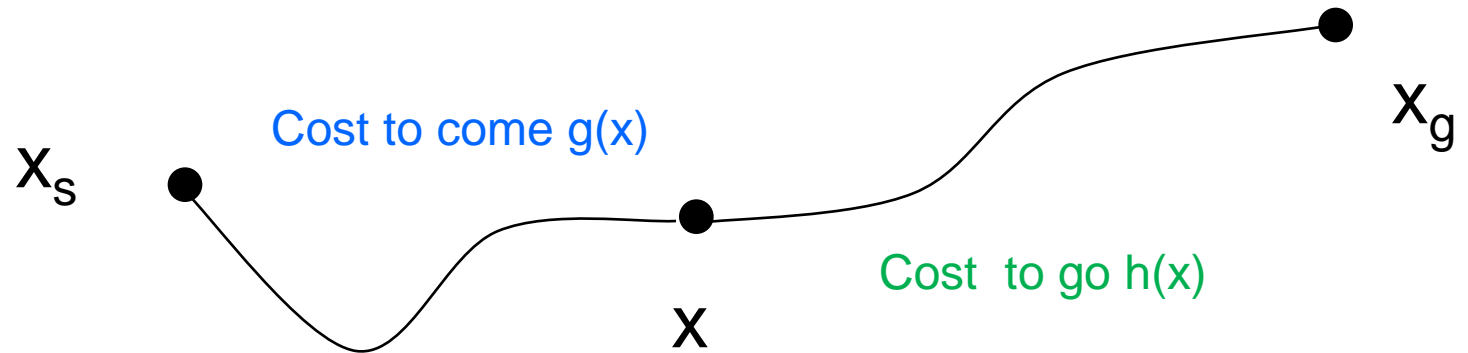
# 10.2.4.1 Optimal Weighted Sequential Planner (Dijkstra's Algorithm)



Dijkstra = (sorta) Grassfire For Nonuniform Edge Cost

# 10.2.4 Weighted Sequential Planning

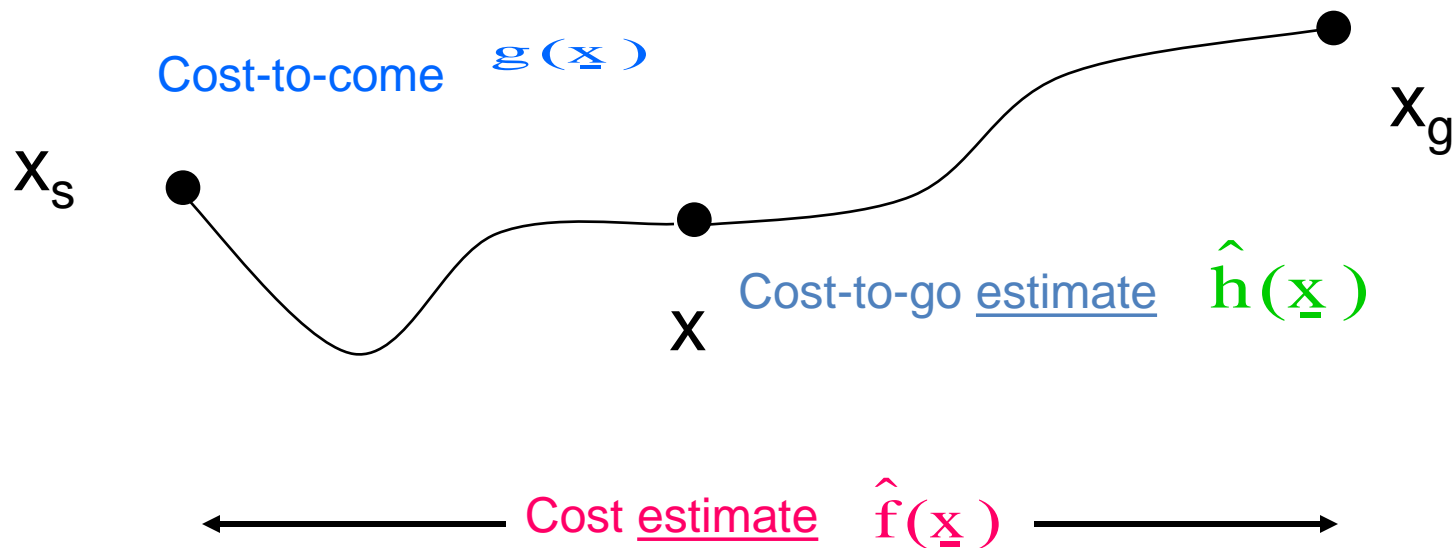
(Reminder: Definitions)



## 10.2.4.2 Heuristic Optimal Weighted Sequential Planner (A\* Algorithm)

- An **estimate** of the cost-to-go makes it possible to be **more efficient** and visit less states than Dijkstra's algorithm
- Now, we **store three values** in each node, called  $f()$ ,  $g()$ , and  $h()$  where:
  - $g(\underline{x})$  is the exact known optimal cost-to-come as it is in Dijkstras algorithm.
  - $\hat{h}(\underline{x})$  is an estimate of the cost-to-go from state to the goal state.
  - $\hat{f}(\underline{x})$  is an estimate (because its based on  $\hat{h}(\underline{x})$ ) of the optimal path cost from the start to the goal through state  $\underline{x}$  computed as follows:

## 10.2.4.2 Heuristic Optimal Weighted Sequential Planner (Achieving Focus in A\* Algorithm)



$$\hat{f}(\underline{x}) = g(\underline{x}) + \hat{h}(\underline{x})$$

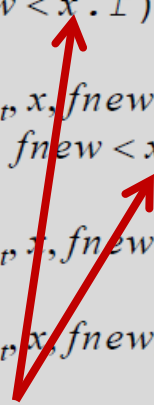
```
00 algorithm calcF(x, g)
01    $f \leftarrow g + \text{calcH}(x)$ 
02   return f
```

## 10.2.4.2 A\* Algorithm: Optimality

- The priority queue is now sorted based on  $f()$ .
  - Amounts to exploring paths **in order of least estimated cost** until the goal is reached.
- When  $h()$  is an underestimate for all  $x$ , the **algorithm is optimal**.
- When the goal is removed from the queue, we know that an underestimate of the cost of all other paths is greater than the actual cost of the present solution, so its optimal.

```
00 algorithm Astar( $x_s, x_g$ )
01  $x_s.g \leftarrow 0$ ;  $x_s.f \leftarrow calcF(x_s, 0)$ 
02  $O.insertSorted(x_s)$ 
03  $x_s.parent \leftarrow null$ 
04 while ( $O \neq \emptyset$ )
05    $x \leftarrow O.removeFirst()$ 
06    $C.insert(x)$ 
07   if ( $x = x_g$ ) return success
08   expandNodeAstar( $x$ )
09 endwhile
10 return failure
```

```
00 algorithm expandNodeAstar( $x$ )
01 for each( $u \in U(x)$ )
02    $x_{next} \leftarrow f(x, u)$ 
03    $g_{new} \leftarrow calcG(x_{next}, x)$ 
04    $f_{new} \leftarrow calcF(x_{next}, g_{new})$ 
05   if ( $x_{next} \in O \ \&\& \ f_{new} < x.f$ )
06      $O.remove(x_{next})$ 
07     addToOAstar( $x_{next}, x, f_{new}, g_{new}$ )
08   else if ( $x_{next} \in C \ \&\& \ f_{new} < x.f$ )
09      $C.remove(x_{next})$ 
10     addToOAstar( $x_{next}, x, f_{new}, g_{new}$ )
11   else
12     addToOAstar( $x_{next}, x, f_{new}, g_{new}$ )
13   endif
14 endfor
15 return
```



**xnext.f**

# 10.2.4.2 Heuristic Optimal Weighted Sequential Planner

```

00 algorithm expandNodeAstar(x)
01 for each(u ∈ U(x))
02   xnext ← f(x, u)
03   gnew ← calcG(xnext, x)
04   fnew ← calcF(xnext, gnew)
05   if(xnext ∈ O && fnew < x.f)
06     O.remove(xnext)
07     addToOAstar(xnext, x, fnew, gnew)
08   else if(xnext ∈ C && fnew < x.f)
09     C.remove(xnext)
10     addToOAstar(xnext, x, fnew, gnew)
11   else
12     addToOAstar(xnext, x, fnew, gnew)
13   endif
14 endfor
15 return
  
```

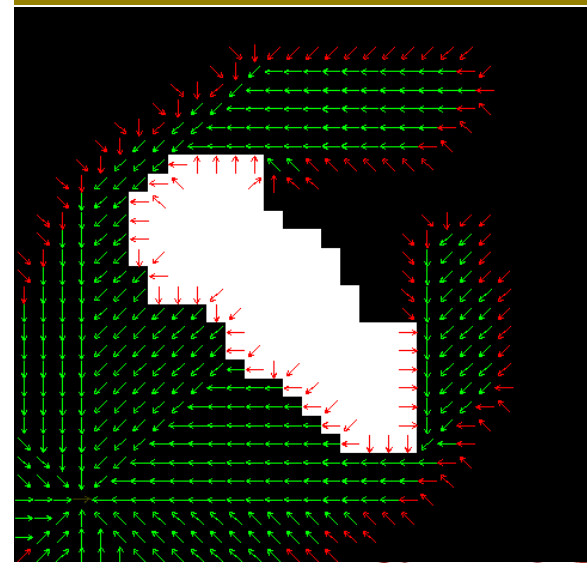
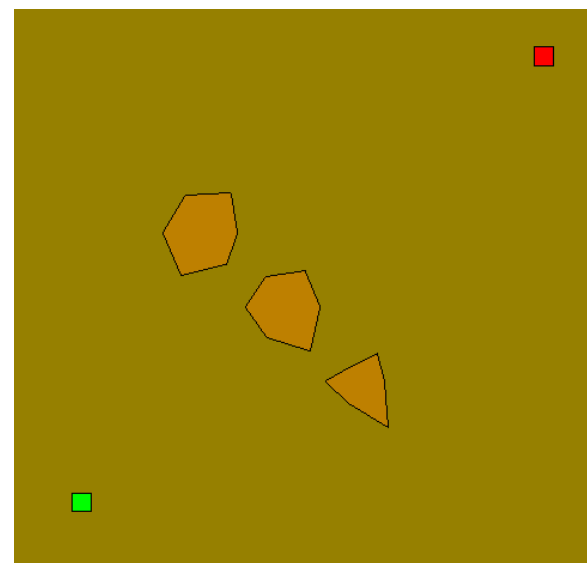
Open Node

Closed Node

New Node

A\* = Heuristic Focused Dijkstra

x<sub>next</sub>.f





## 10.2.4.2 Heuristic Optimal Weighted Sequential Planner (Re Adding Nodes to “Open”)

- A handy routine to save writing it three times.

```
00 algorithm addToOAstar ( $x, x_{parent}, f, g$ )  
01  $x.g \leftarrow g$   
02  $x.f \leftarrow f$   
03  $O.insertSorted(x, f)$   
04  $x.parent \leftarrow x_{parent}$   
05 return
```

## 10.2.4.2 Heuristic Optimal Weighted Sequential Planner (A\* Facts)

- “Admissability” (implies optimality)
  - Let  $h^*(x)$  mean the true optimal cost to the goal from state  $x$ .
  - $h(x)$  is “admissible” if it is always an underestimate of the true cost to the goal.
    - $h(x) \leq h^*(x)$  always
- Not supposed to call the algorithm A\* if  $h(x)$  is not admissible. (Call it simply A)

## 10.2.4.2 Heuristic Optimal Weighted Sequential Planner (A\* Facts)

- “Informed” (relates to efficiency)
  - $h_1(x)$  is **more informed** than  $h_2(x)$  if:
    - $h_1(x) > h_2(x)$  and ...
    - both are admissible
- A search based on  $h_1(x)$  will open a subset of the nodes opened using  $h_2(x)$
- Hence  $h_1(x)$  is more efficient.



Its just like the  
“THE PRICE IS RIGHT!”

## 10.2.4.5 Monotonicity of Total Cost for Consistent Heuristics

- $h(x)$  is “monotone” if it satisfies the triangle inequality:

- for any arc  $(n, n')$  we have

- $h(n) \leq h(n') + c(n, n')$

Going straight to  $g$  from  $n$  must be cheaper than going from  $n$  to  $n'$  to  $g$ .

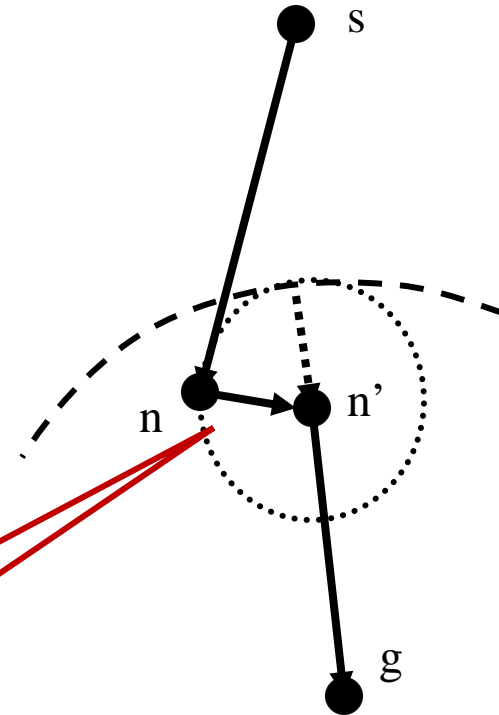
- Any monotone heuristic is admissible.
- For the sequence of nodes  $n_1, n_2, \dots$  Opened under these circumstances, we have:

- $f(n_1) \leq f(n_2) \leq \dots$

- The  $f$  values grow monotonically.

- A node will never be moved from closed to open.

Think of:  
 $n$  = parent  
 $n'$  = child



# Outline

- 10.2 Representation and Search for Global Motion Planning
  - 10.2.1 Sequential Motion Planning
  - 10.2.2 Big Ideas in Optimization and Search
  - 10.2.3 Uniform Cost Sequential Planning Algorithms
  - 10.2.4 Wei *skip* ↓ Sequential Planning
  - 10.2.5 Representation For Sequential Motion Planning
  - Summary

# Outline

- 10.2 Representation and Search for Global Motion Planning
  - 10.2.1 Sequential Motion Planning
  - 10.2.2 Big Ideas in Optimization and Search
  - 10.2.3 Uniform Cost Sequential Planning Algorithms
  - 10.2.4 Weighted Sequential Planning
  - 10.2.5 Representation For Sequential Motion Planning
  - Summary

# Summary

- When vehicles are not omnidirectional, even planning without obstacles is hard.
- Planning in the continuum is not usually attempted:
  - when there are significant obstacles or
  - interesting cost fields.
  - Instead, convert the problem to an SDP
- For SDPs, techniques of substantial elegance exist.
  - Sorting a priority queue
  - Heuristics