Real-Time Planning in Dynamic and Partially Known Domains

Maxim Likhachev
University of Pennsylvania
maximl@seas.upenn.edu

Sven Koenig
University of Southern California
skoenig@usc.edu

Warning!

- We try to make everything easy to understand.
- We often do not mention crucial details.
- We use both 4- and 8-neighbor grids.
- We invite you to ask questions!

Warning!

- We use robotics to illustrate the planning techniques because
  - incomplete information and uncertainty are important in robotics
  - domains from robotics are easy to understand, and
  - the behavior of planning techniques is easy to visualize.
- However, the planning techniques also apply to a variety of other domains, including more “symbolic” ones.

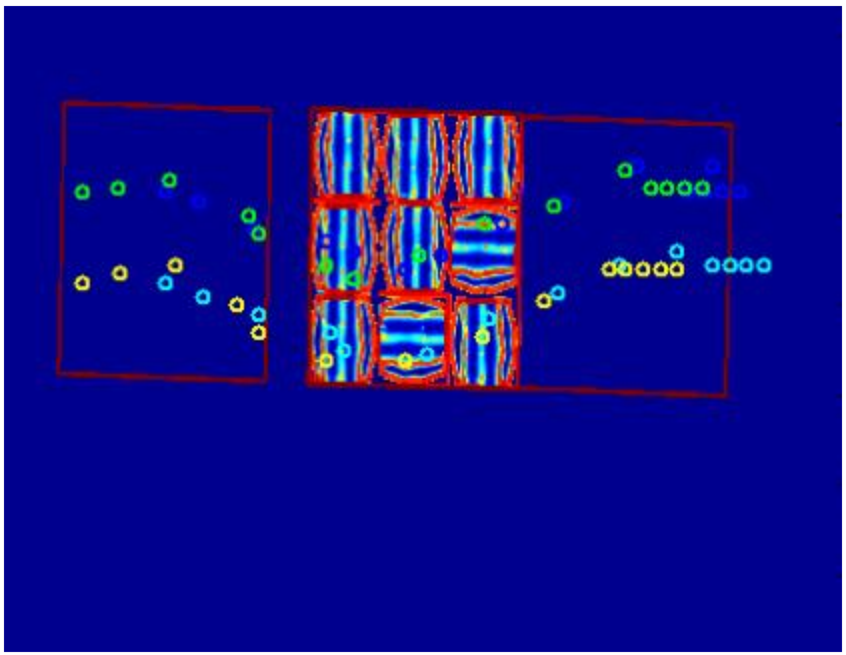
Real-time Planning in Dynamic and Partially-known Domains

- Challenges
  - complexity/size (high-dim., expensive to compute costs, etc.)

Real-time Planning in Dynamic and Partially-known Domains

- Challenges
  - complexity/size (high-dim., expensive to compute costs, etc.)

planning in 8D (<x,y> for each foothold) using R*



Real-time Planning in Dynamic and Partially-known Domains

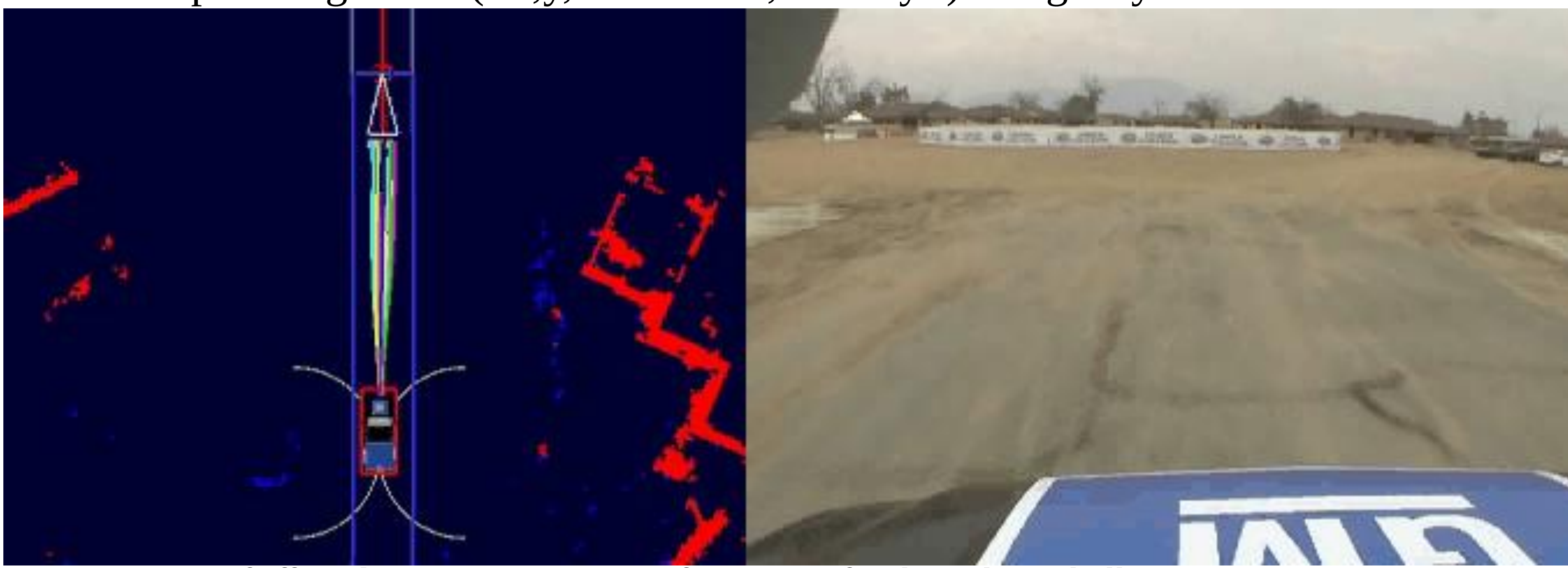
- Challenges
  - complexity/size (high-dim., expensive to compute costs, etc.)
  - severe time constraints (e.g., tens of msecs to few seconds)

Real-time Planning in Dynamic and Partially-known Domains

Maxim

- Challenges
  - complexity/size (high-dim., expensive to compute costs, etc.)
  - severe time constraints (e.g., tens of msecs to few seconds)
  - robustness to uncertainties in execution, sensing, environment

planning in 4D (<x,y,orientation,velocity>) using Anytime D*



part of efforts by Tartanracing team from CMU for the Urban Challenge 2007 race

Real-time Planning in Dynamic and Partially-known Domains

- Challenges
  - complexity/size (high-dim., expensive to compute costs, etc.)
  - severe time constraints (e.g., tens of msecs to few seconds)
  - robustness to uncertainties in execution, sensing, environment
  - generality of approaches
  - theoretical guarantees
  - simplicity

Real-time Planning in Dynamic and Partially-known Domains

Maxim

- Challenges
  - complexity/size (high-dim., expensive to compute costs, etc.)
  - severe time constraints (e.g., tens of msecs to few seconds)
  - robustness to uncertainties in execution, sensing, environment

  - generality of approaches
  - theoretical guarantees
  - simplicity

usually satisfied by graph searches such as A*

ability to find some solution fast

ability to improve the solution before and during execution

ability to re-use search results

ability to plan under uncertainty

This talk!

Real-time Planning in Dynamic and Partially-known Domains

Common theme in this talk:

- Planning with a series of (efficient) graph searches
- Planning with variants of A* searches

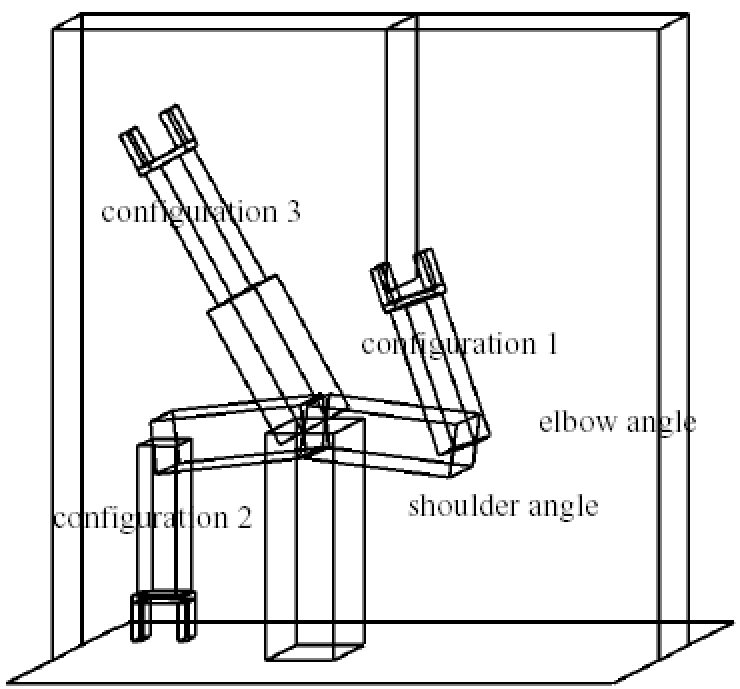
Table of Contents

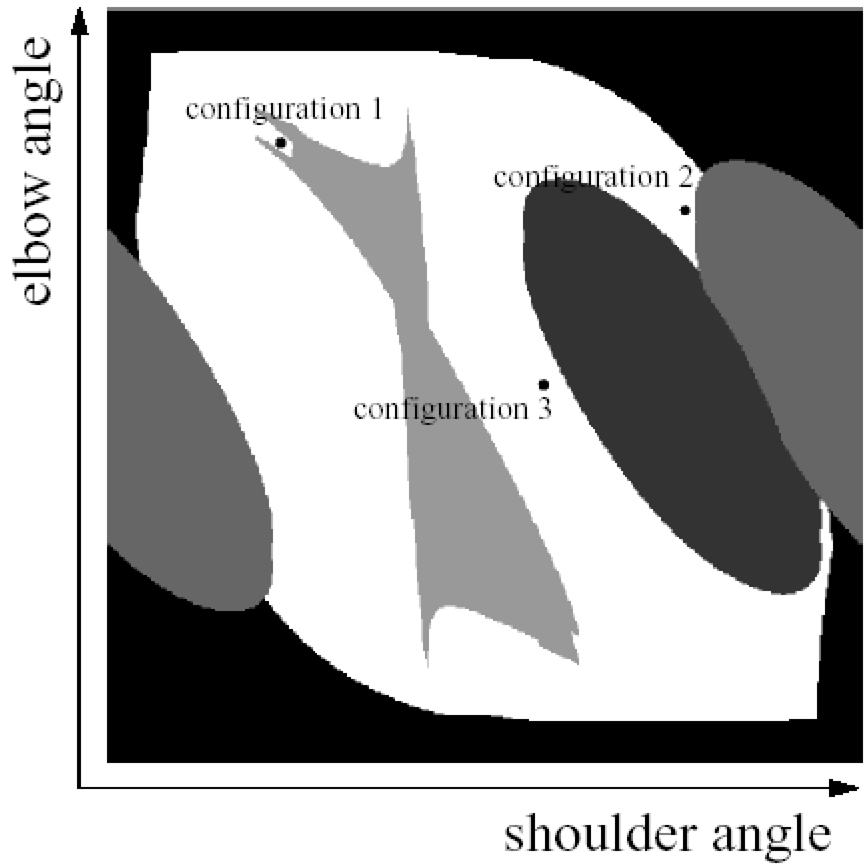
- Modeling Planning Domains
  - Graphs, MDPs
- Planning Problems and Strategies
  - Localization, Mapping, Navigation in Unknown Terrain
  - Agent-Centered Search, Assumptive Planning
- Efficient Implementations of Planning Strategies
  - Incremental Heuristic Search

15 Minute Break

  - Real-Time Heuristic Search
  - Planning with Preferences on Uncertainty
  - Planning with Varying Abstractions

Work vs Configuration Space





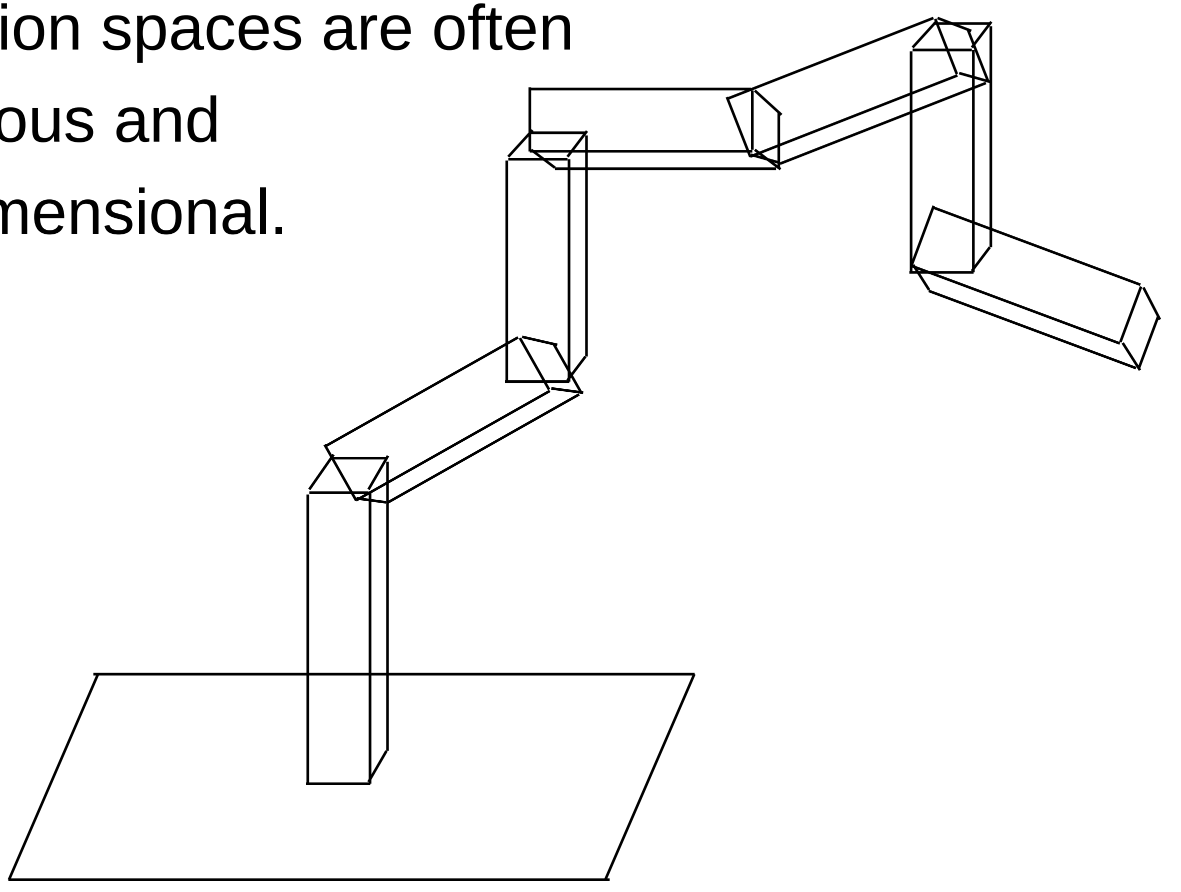
[from Stuart Russell and Peter Norvig]

work space

configuration space

Work vs Configuration Space

- Configuration spaces are often
  - continuous and
  - high-dimensional.

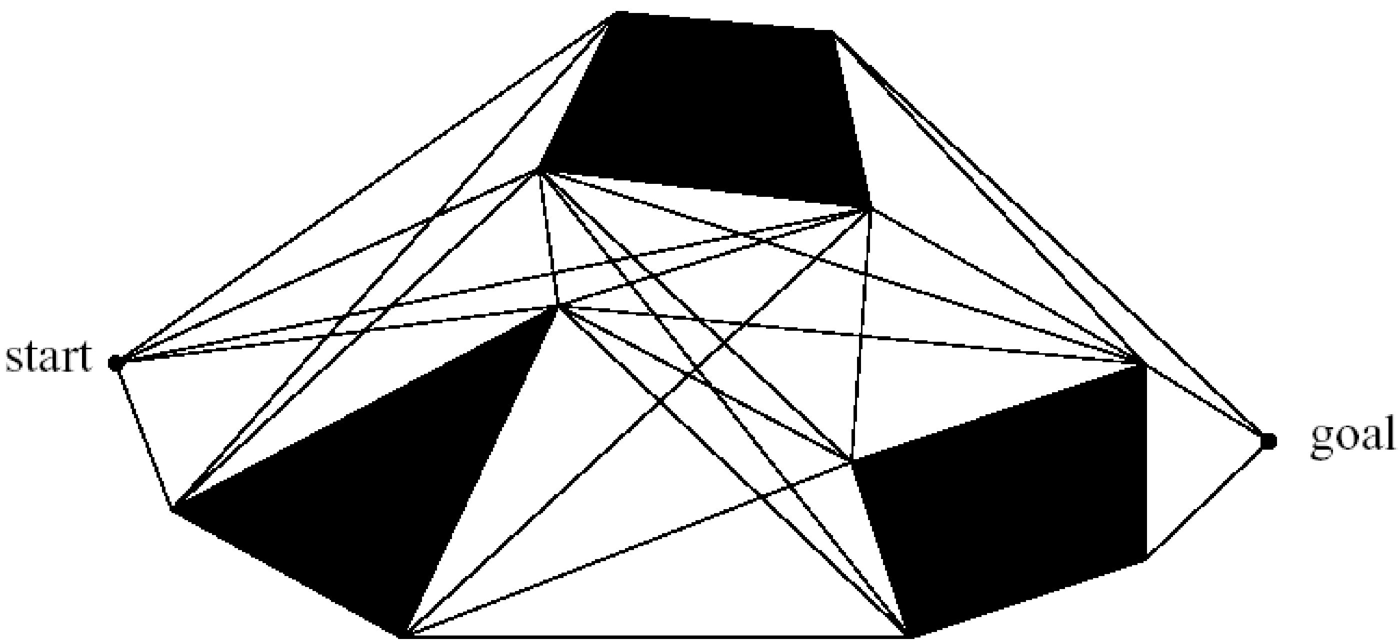


Modeling Planning Domains

- Deterministic Models – Graphs
  - Skeletonization Methods (Roadmaps)
  - Cell Decomposition Methods
- Searching Graphs
  - A*
  - Weighted A*
- Nondeterministic Models – MDPs
- Searching MDPs

Discretizing Configuration Space

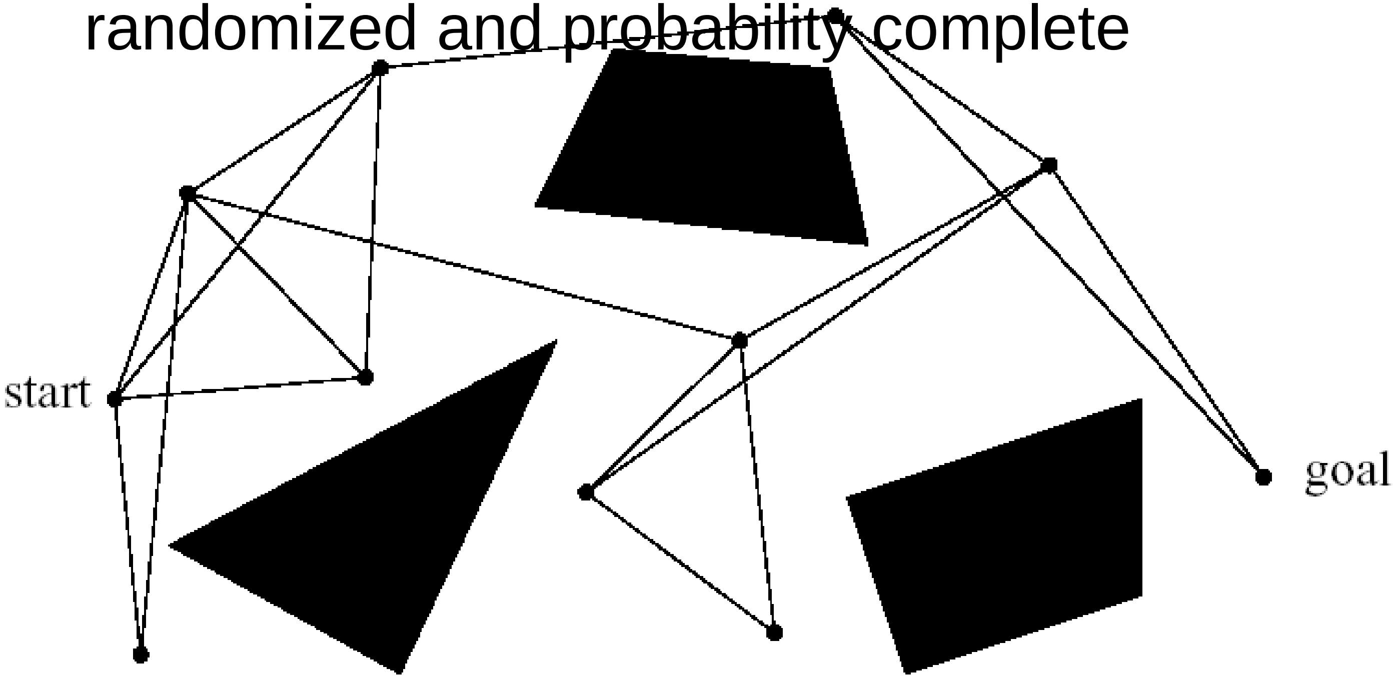
- Skeletonization methods



visibility graph

Discretizing Configuration Space

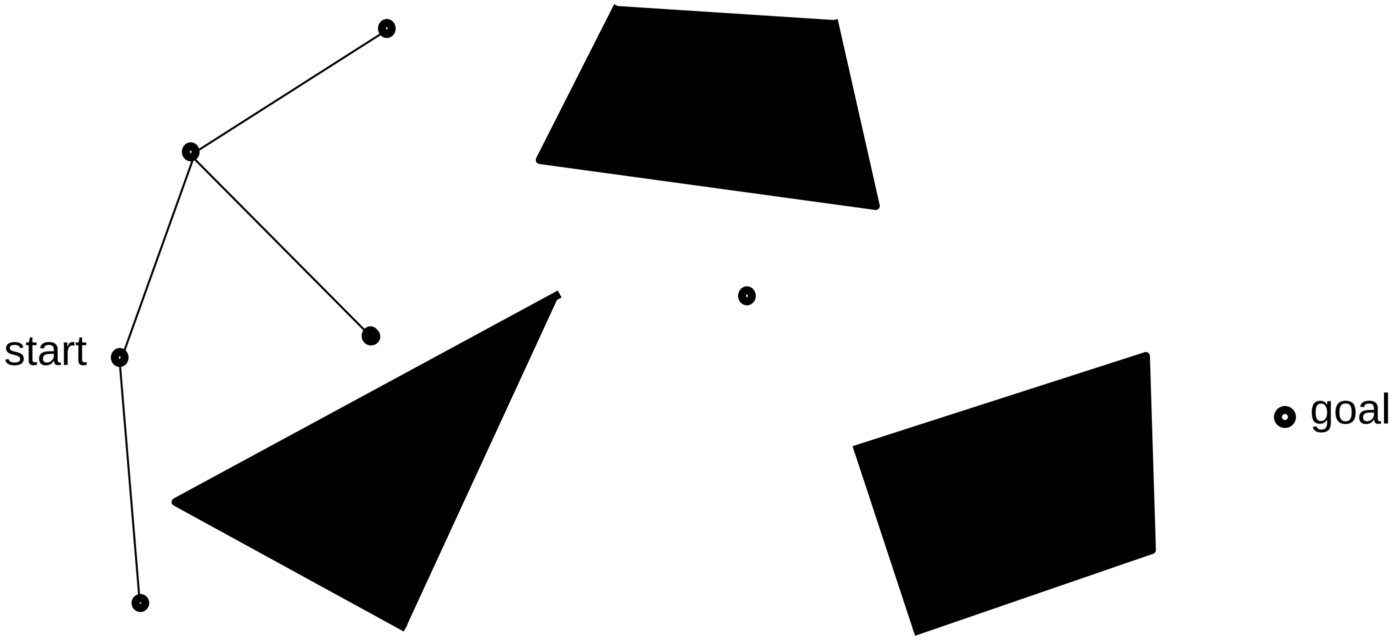
- Skeletonization methods:
  randomized and probability complete



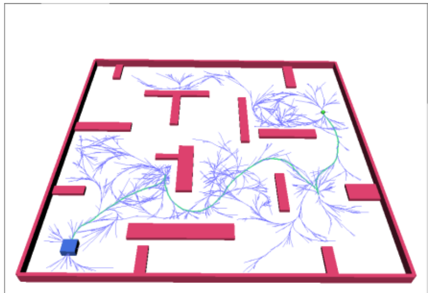
roadmap using random points [Kavraki et al, 1994]

Discretizing Configuration Space

- Skeletonization methods:
  randomized and probability complete



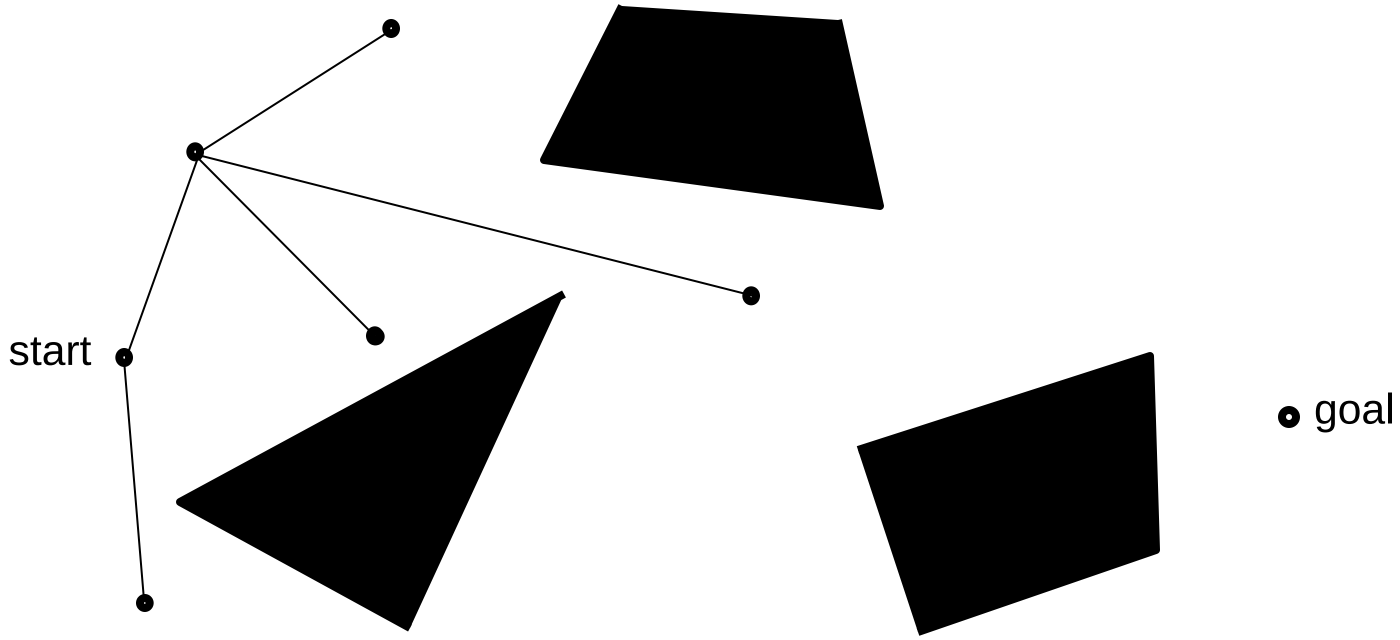
roadmaps using RRTs [LaValle, 1998]



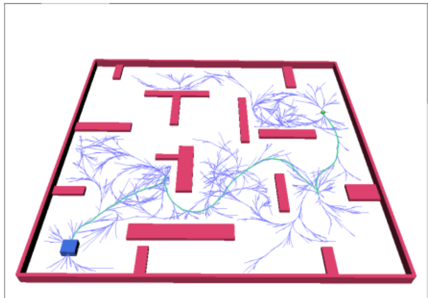
[from Steve LaValle]

Discretizing Configuration Space

- Skeletonization methods:
  randomized and probability complete



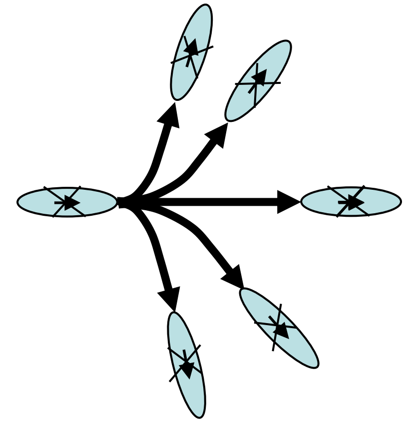
roadmaps using RRTs [LaValle, 1998]

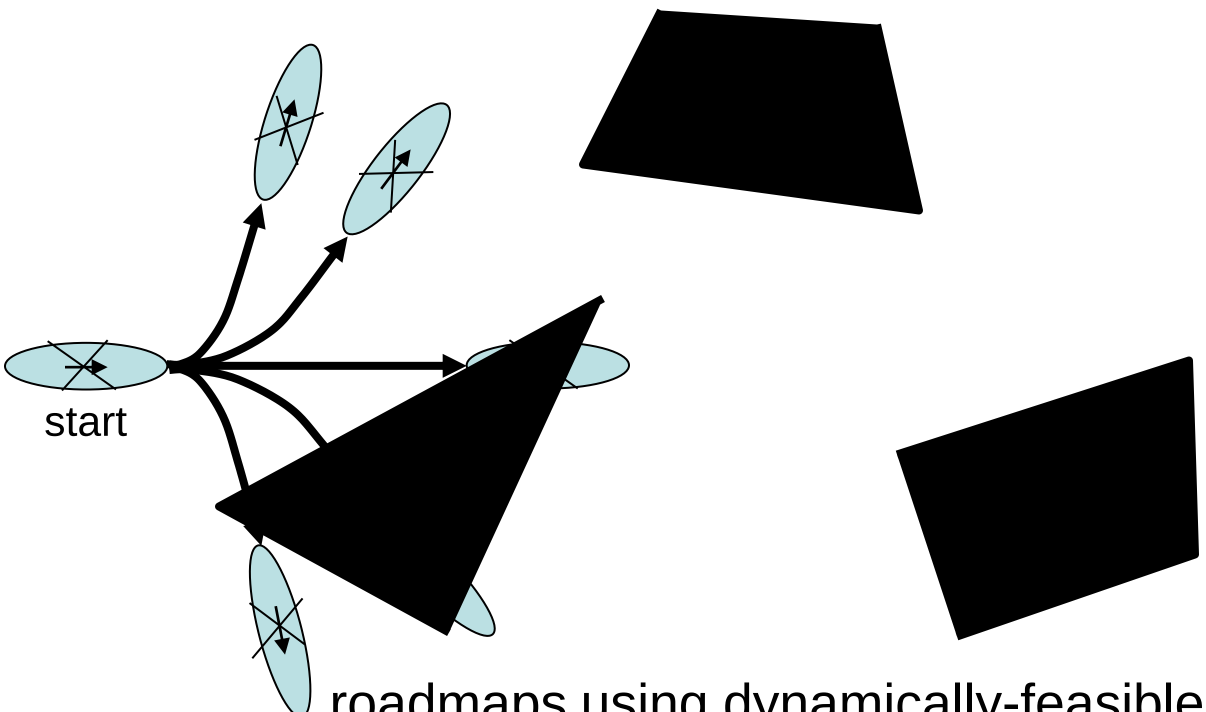


[from Steve LaValle]

Discretizing Configuration Space

- Skeletonization methods:
  randomized and probability complete

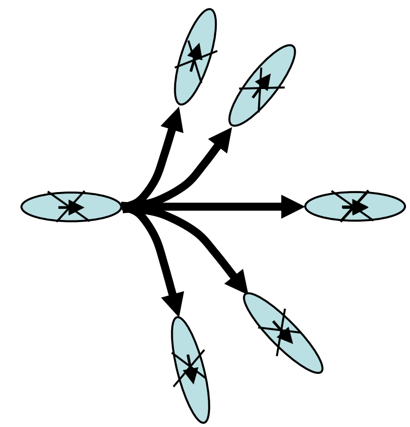


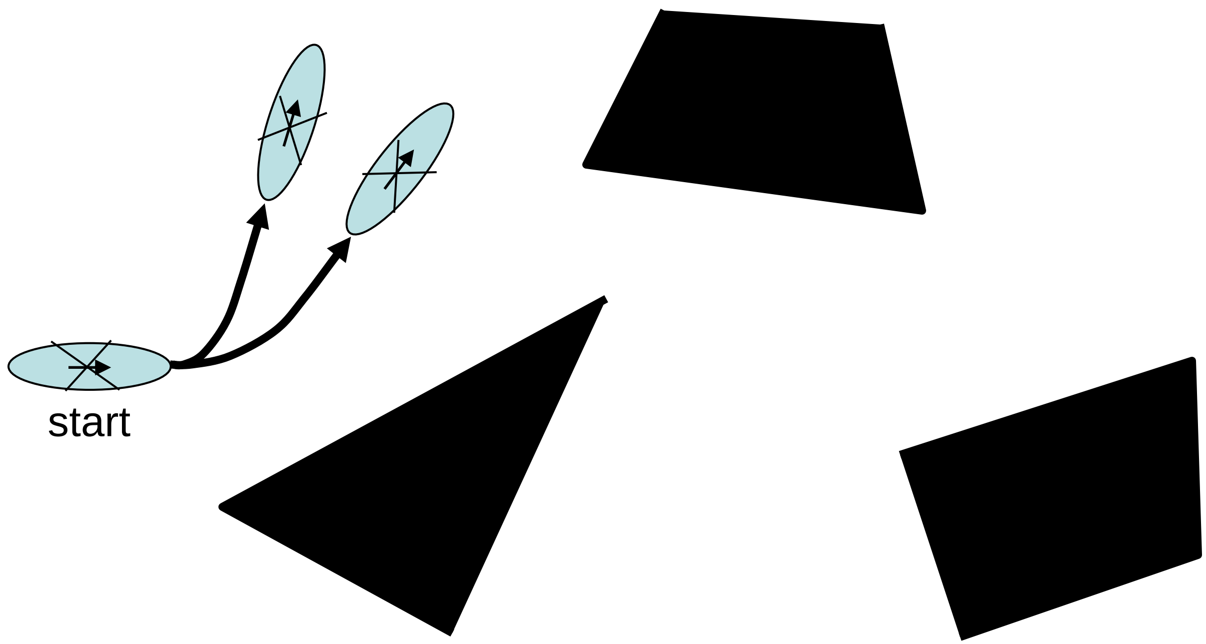


roadmaps using dynamically-feasible trajectories

Discretizing Configuration Space

- Skeletonization methods:
  randomized and probability complete

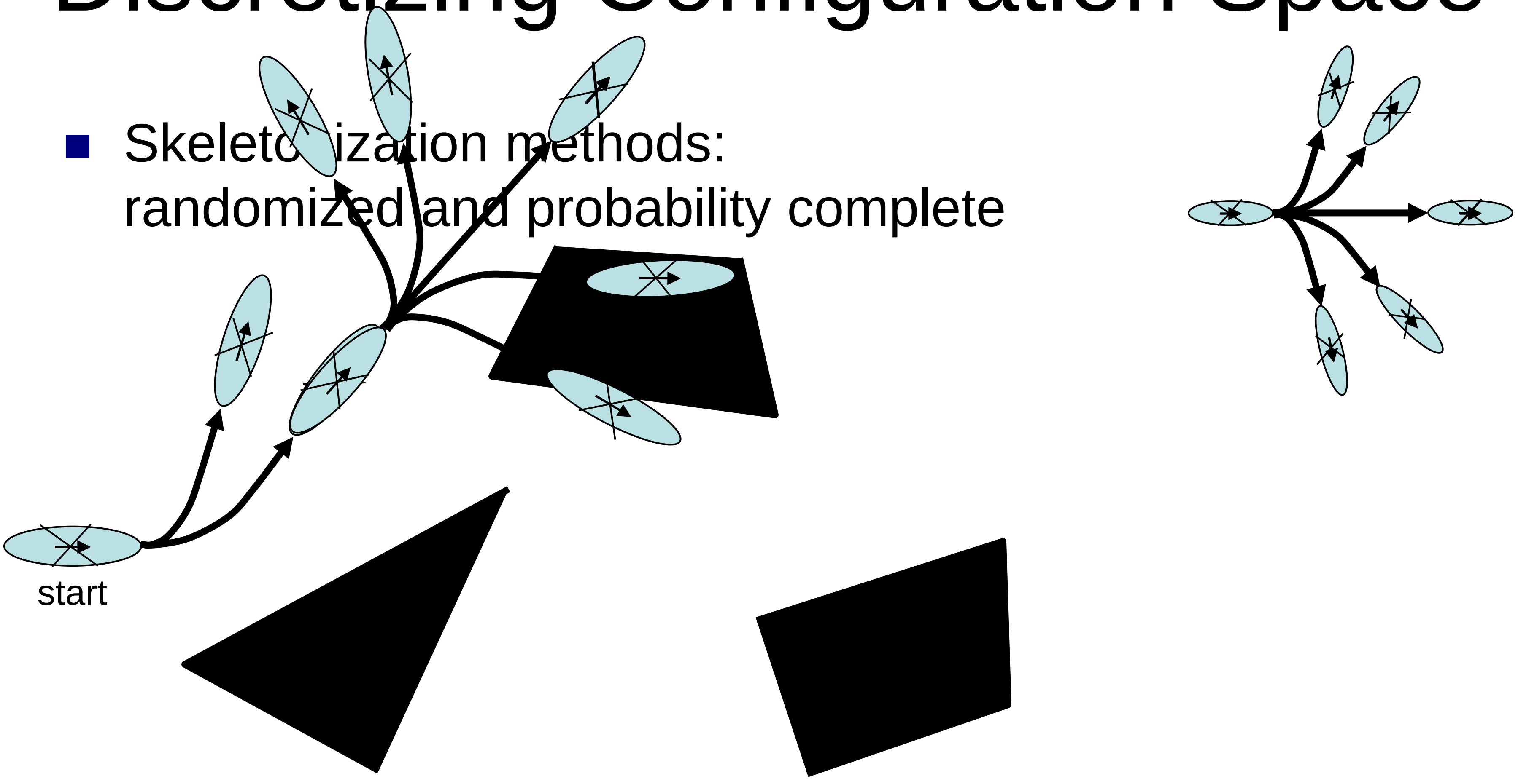




roadmaps using dynamically-feasible trajectories

Discretizing Configuration Space

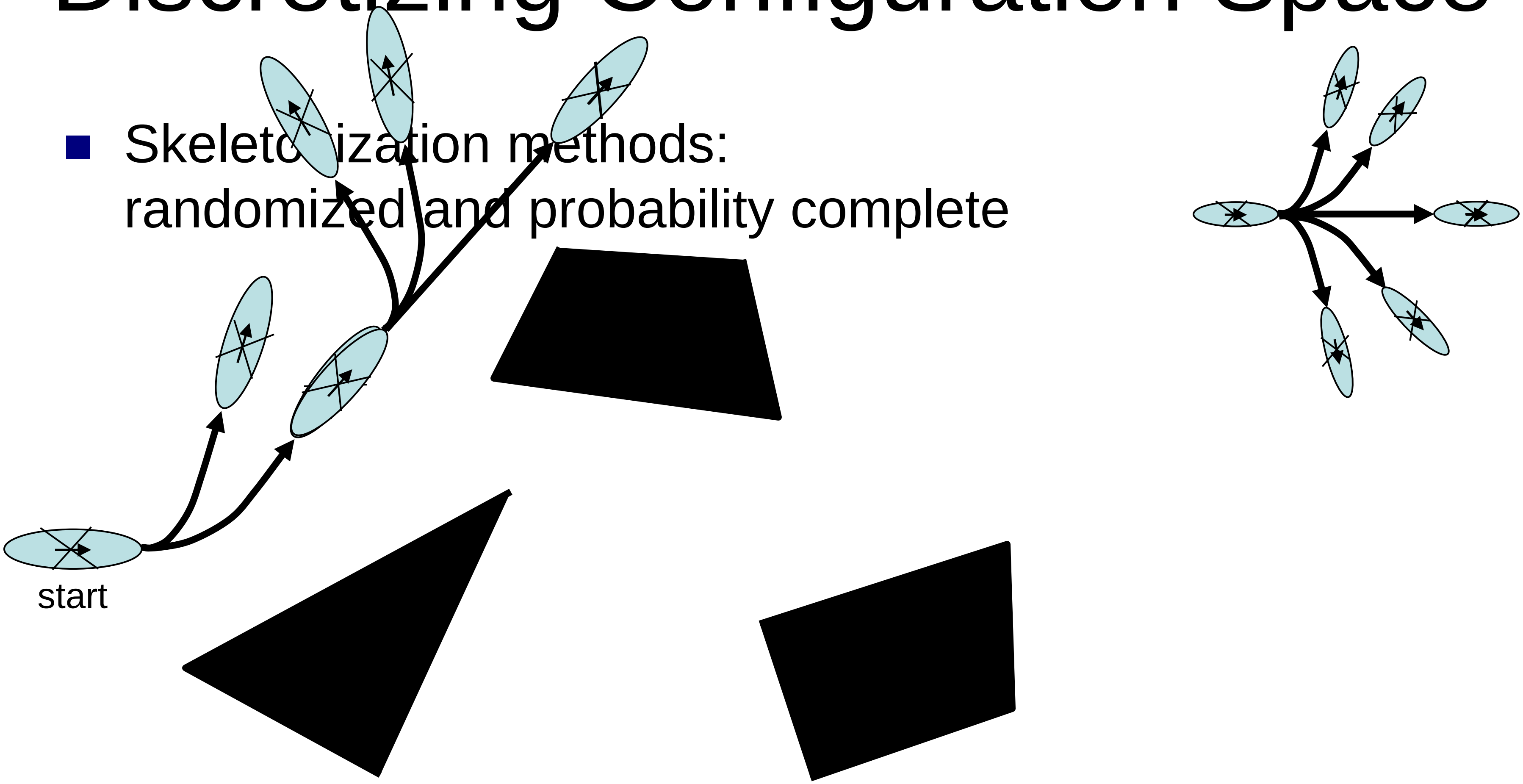
- Skeletonization methods:
  randomized and probability complete



roadmaps using dynamically-feasible trajectories

Discretizing Configuration Space

- Skeletonization methods:
  randomized and probability complete



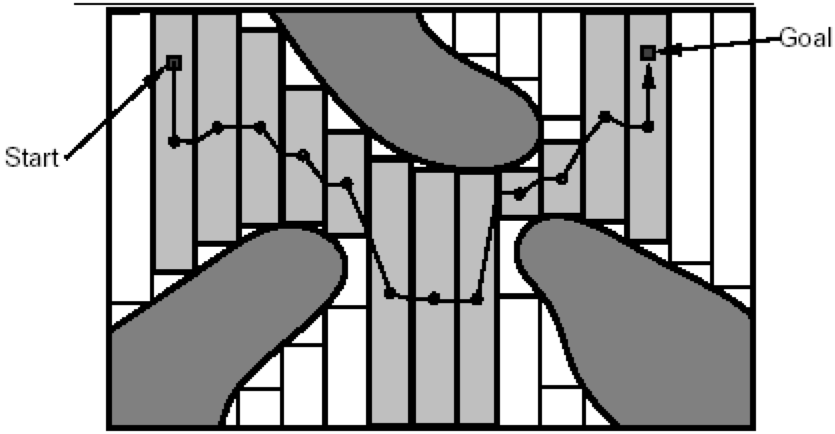
roadmaps using dynamically-feasible trajectories

Modeling Planning Domains

- Deterministic Models – Graphs
  - Skeletonization Methods (Roadmaps)
  - Cell Decomposition Methods
- Searching Graphs
  - A*
  - Weighted A*
- Nondeterministic Models – MDPs
- Searching MDPs

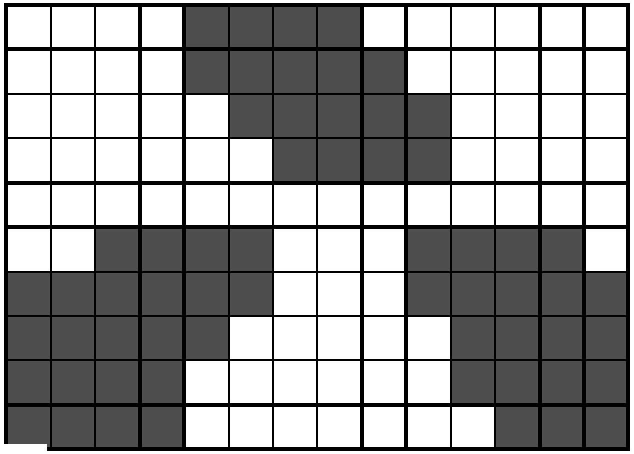
Discretizing Configuration Space

- Cell decomposition methods:
  systematic and resolution complete



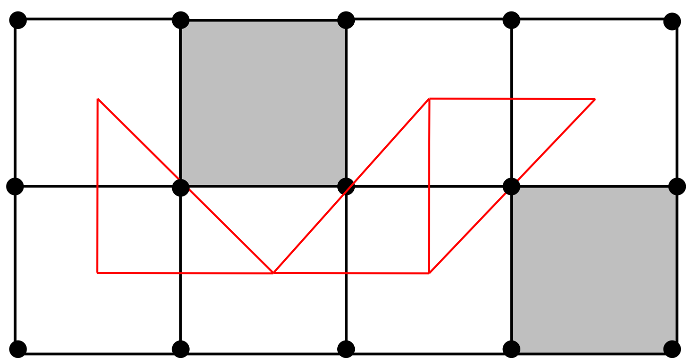
[from Stuart Russell and Peter Norvig]

vertical strips

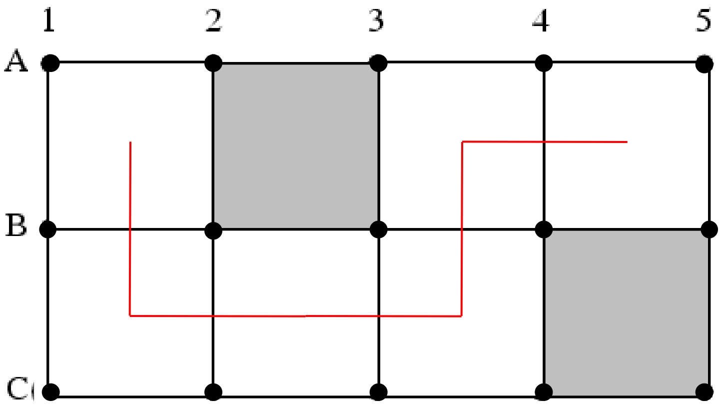


grid

Discretizing Configuration Space



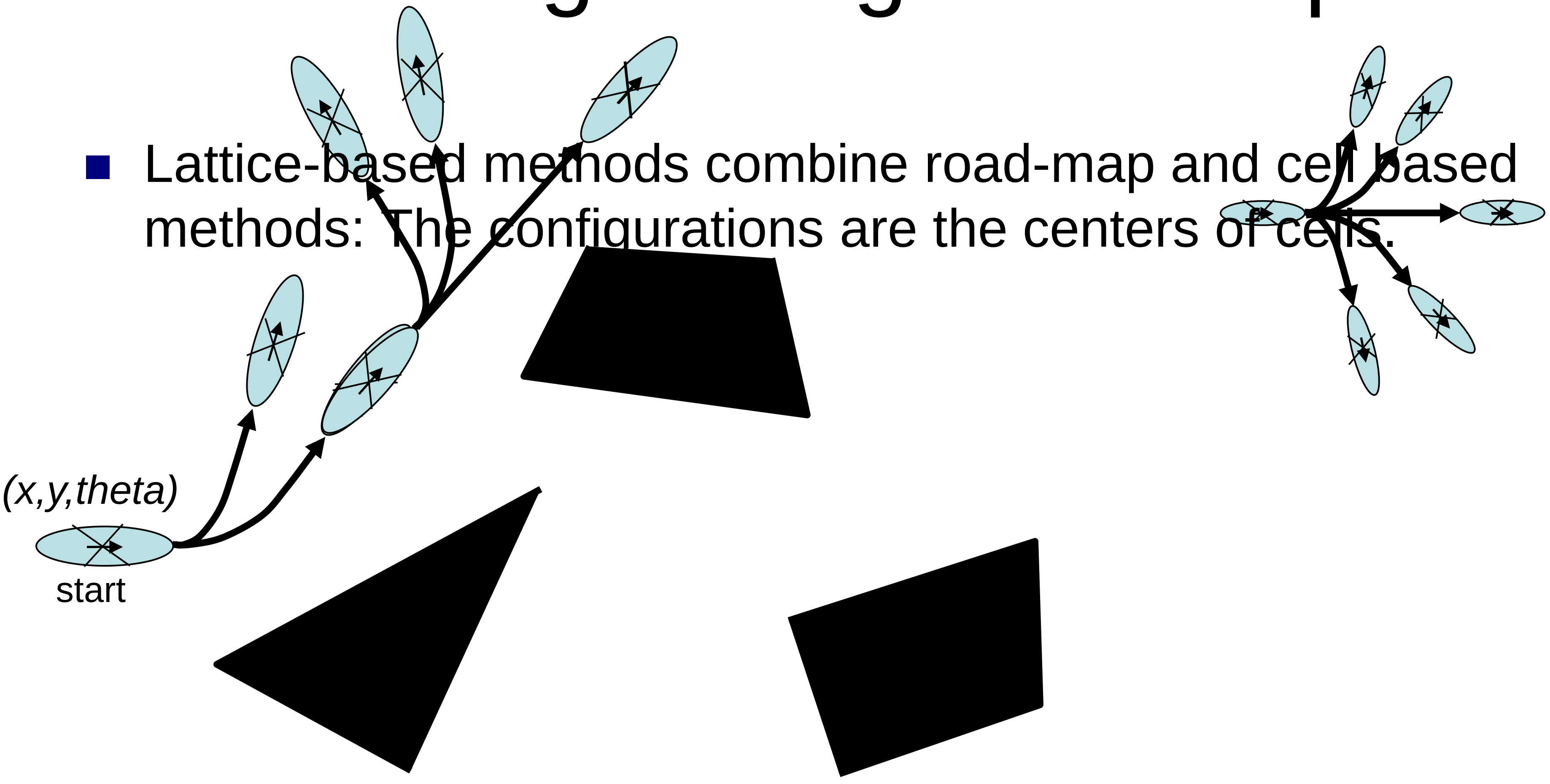
8-neighbor grid



4-neighbor grid

Discretizing Configuration Space

- Lattice-based methods combine road-map and cell based methods: The configurations are the centers of cells.

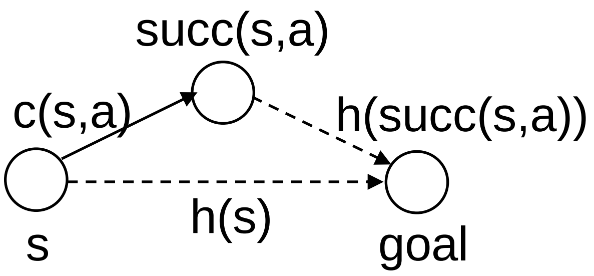


Modeling Planning Domains

- Deterministic Models – Graphs
  - Skeletonization Methods (Roadmaps)
  - Cell Decomposition Methods
- Searching Graphs
  - A*
  - Weighted A*
- Nondeterministic Models – MDPs
- Searching MDPs

A*

- A* [Hart, Nilsson and Raphael, 1968] uses user-supplied h-values to focus its search.
- The h-values approximate the goal distances.
- We always assume that the h-values are consistent!
- The h-values h(s) are consistent iff they satisfy the triangle inequality:
  h(s) = 0 if s is the goal and
  h(s) ≤ c(s,a) + h(succ(s,a)) otherwise.



- Consistent h-values are admissible.
- The h-values h(s) are admissible iff they do not overestimate the goal distances.

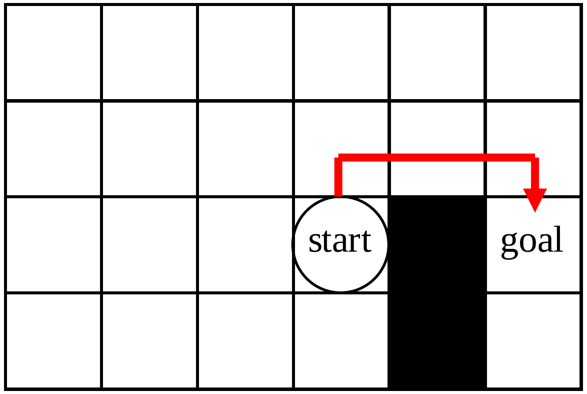
A*

(Forward) A*

1. Create a search tree that contains only the start.
2. Pick a generated but not yet expanded state s with the smallest f-value.
3. If state s is the goal then stop.
4. Expand state s.
5. Go to 2.

A*

- Search problem with uniform cost



4-neighbor grid

A*

- Possible consistent h-values

7	6	5	4	3	2
6	5	4	3	2	1
5	4	3	2	1	0
6	5	4	3	2	1

Manhattan Distance

5	4	3	2	2	2
5	4	3	2	1	1
5	4	3	2	1	0
5	4	3	2	1	1

Octile Distance

0	0	0	0	0	0
0	0	0	0	0	0
0	0	0	0	0	0
0	0	0	0	0	0

Zero h-values

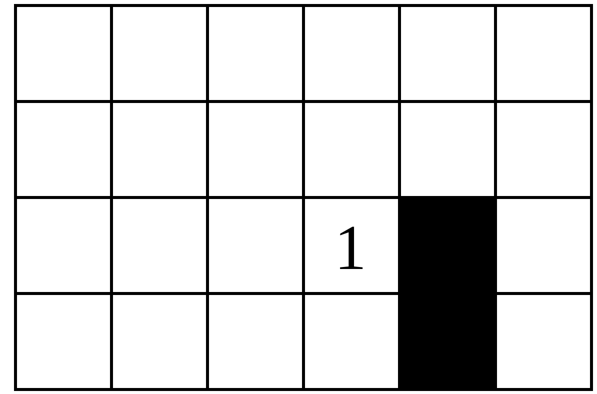
more informed (dominating)

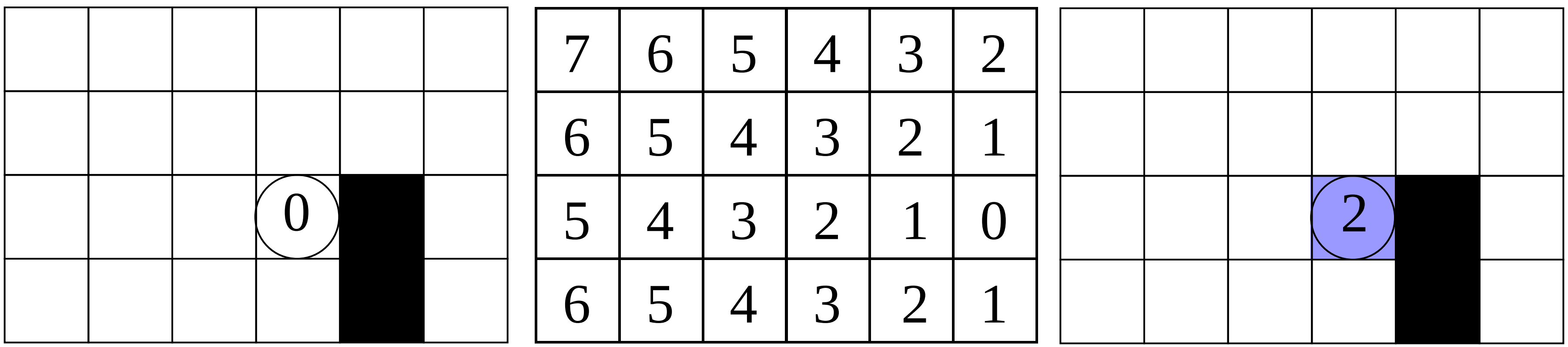
4-neighbor grid

A*

- First iteration of A*

order of expansions





g-values + h-values = f-values

cost of the shortest path in the search tree from the start to the given state

generated but not expanded state (OPEN list)

expanded state (CLOSED list)

4-neighbor grid

A*

order of expansions

			1		
			2		

- Second iteration of A*

g-values

			1		
		1	0		
			1		

+

h-values

7	6	5	4	3	2
6	5	4	3	2	1
5	4	3	2	1	0
6	5	4	3	2	1

=

f-values

			4		
		4	2		
			4		

cost of the shortest path in the search tree from the start to the given state

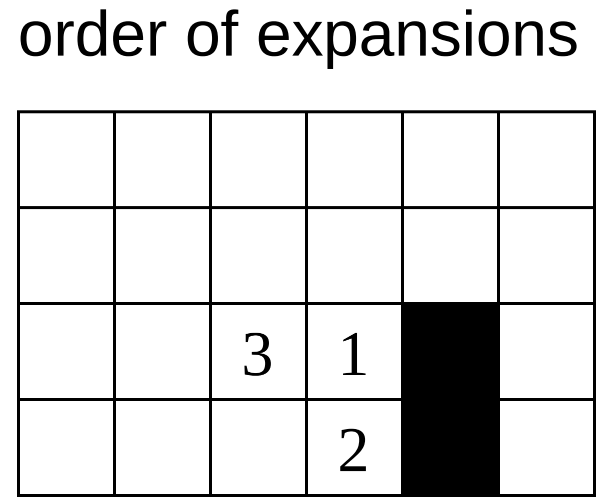
generated but not expanded state (OPEN list)

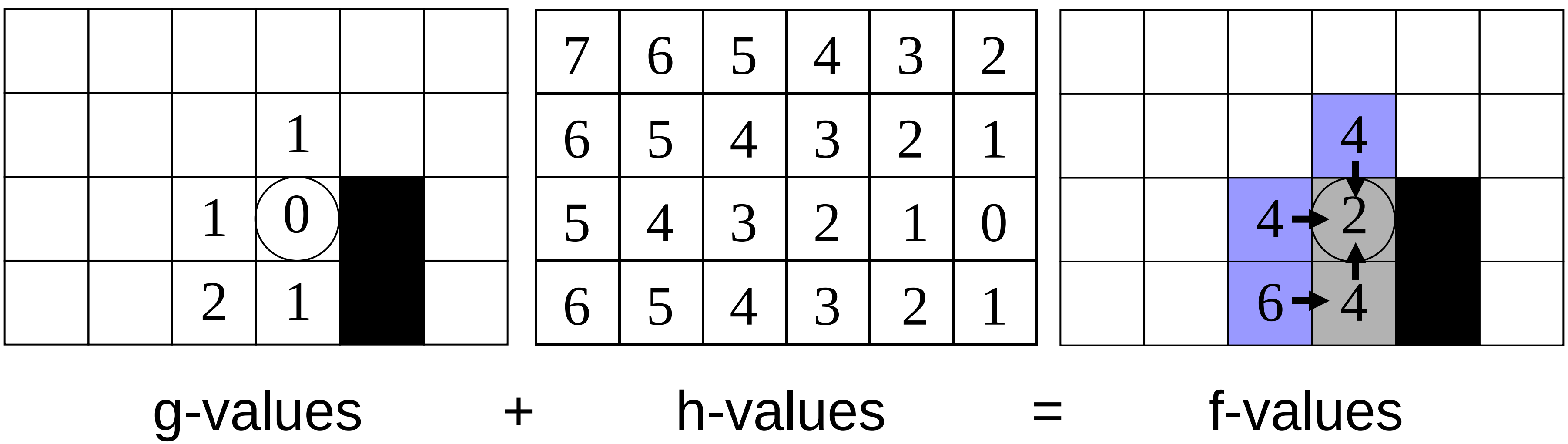
expanded state (CLOSED list)

4-neighbor grid

A*

- Third iteration of A*





cost of the shortest path in the search tree from the start to the given state

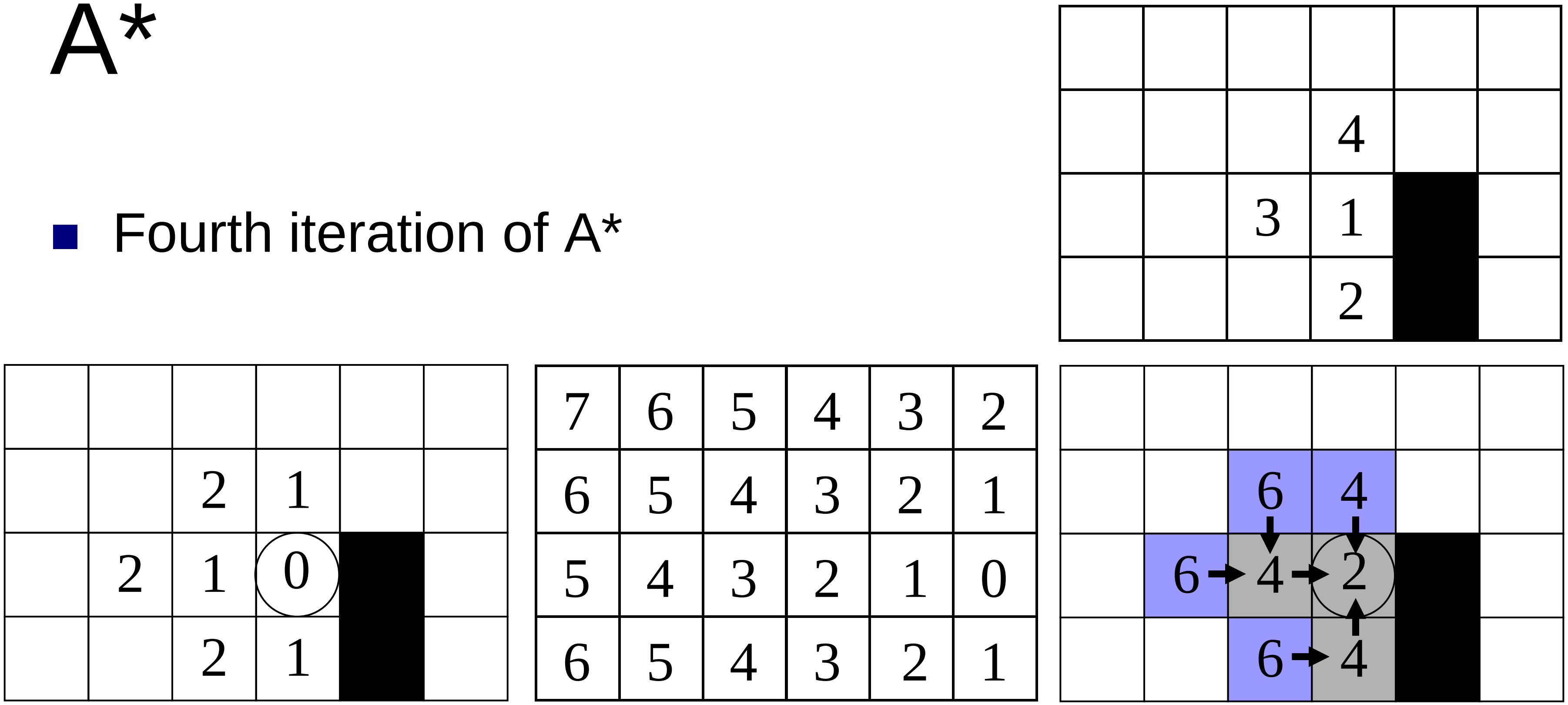
generated but not expanded state (OPEN list)
expanded state (CLOSED list)

4-neighbor grid

A*

- Fourth iteration of A*

order of expansions



g-values + h-values = f-values

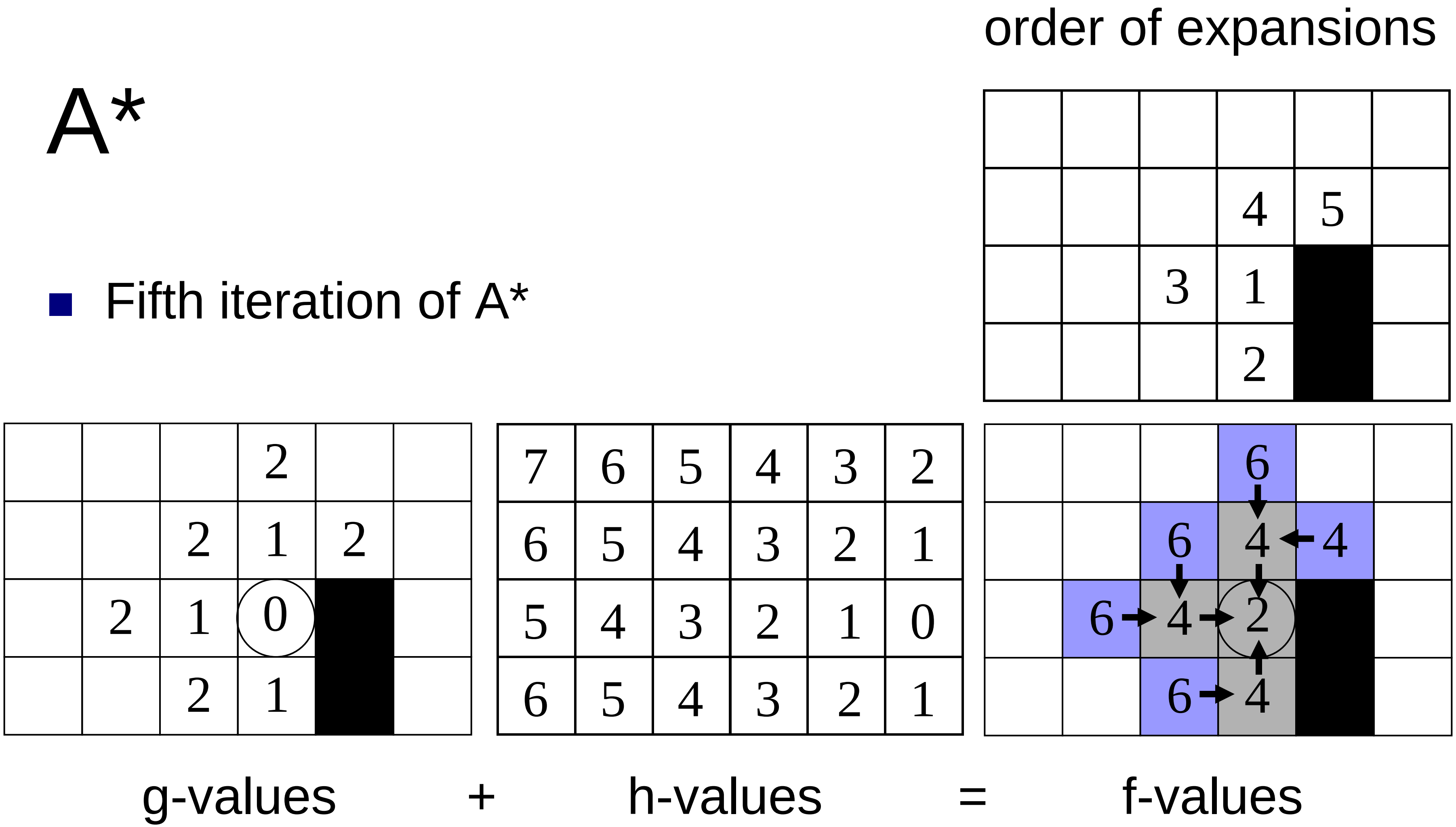
cost of the shortest path in the search tree from the start to the given state

- generated but not expanded state (OPEN list)
- expanded state (CLOSED list)

4-neighbor grid

A*

- Fifth iteration of A*



cost of the shortest path in the search tree from the start to the given state

generated but not expanded state (OPEN list)

expanded state (CLOSED list)

4-neighbor grid

A*

order of expansions

			4	5	6
		3	1		
			2		

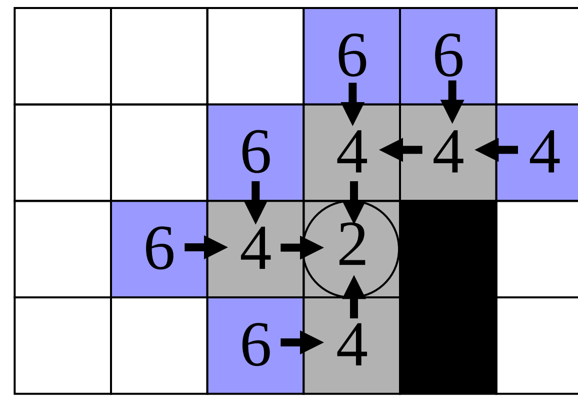
- Sixth iteration of A*

			2	3	
		2	1	2	3
	2	1	0		
		2	1		

g-values +

7	6	5	4	3	2
6	5	4	3	2	1
5	4	3	2	1	0
6	5	4	3	2	1

h-values =



f-values

cost of the shortest path in the search tree from the start to the given state

generated but not expanded state (OPEN list)
expanded state (CLOSED list)

4-neighbor grid

A*

- Seventh and last iteration of A*

order of expansions

			4	5	6
		3	1		(7)
			2		

g-values

			2	3	4
		2	1	2	3
	2	1	0		4
		2	1		

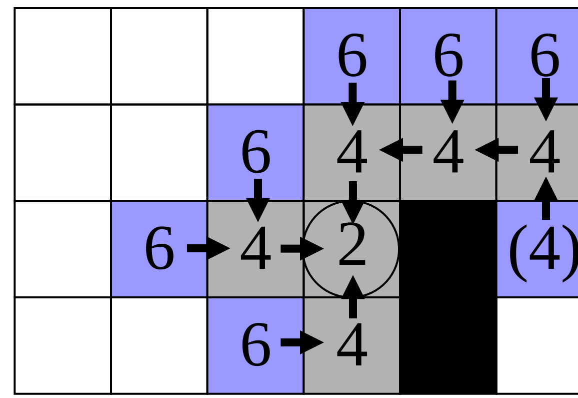
\+

h-values

7	6	5	4	3	2
6	5	4	3	2	1
5	4	3	2	1	0
6	5	4	3	2	1

=

f-values



cost of the shortest path in the search tree from the start to the given state

generated but not expanded state (OPEN list)

expanded state (CLOSED list)

4-neighbor grid

A*

Uniform-cost search
Breadth-first search

7	6	5	4	3	2
6	5	4	3	2	1
5	4	3	2	1	0
6	5	4	3	2	1

Manhattan Distance

5	4	3	2	2	2
5	4	3	2	1	1
5	4	3	2	1	0
5	4	3	2	1	1

Octile Distance

0	0	0	0	0	0
0	0	0	0	0	0
0	0	0	0	0	0
0	0	0	0	0	0

Zero h-values

			4	5	6
		3	1		(7)
			2		

			6		
			3	4	7
		5	1		(8)
			2		

	18	13	8	14	19
17	12	7	4	9	15
11	6	3	1		(20)
16	10	5	2		

← more informed (dominating)

4-neighbor grid

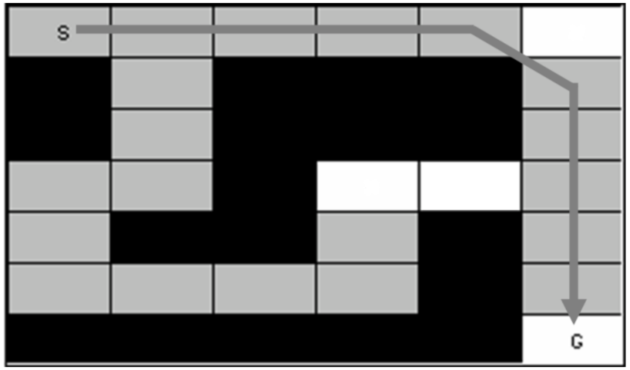
Modeling Planning Domains

- Deterministic Models – Graphs
  - Skeletonization Methods (Roadmaps)
  - Cell Decomposition Methods
- Searching Graphs
  - A*
  - Weighted A*
- Nondeterministic Models – MDPs
- Searching MDPs

Weighted A*

A*

f(s) = g(s) + h(s)



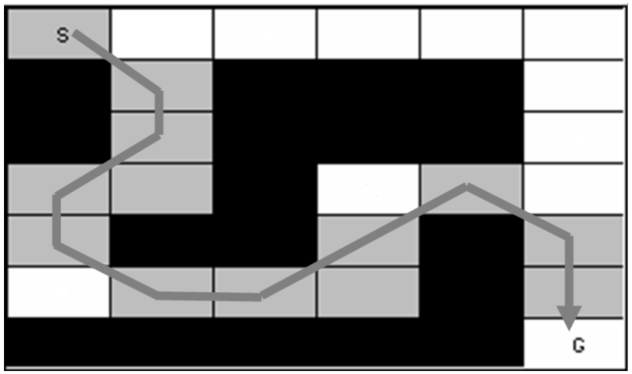
(w = 1.0)

20 expansions

10 movements

Weighted A* [Pohl, 1970]

f(s) = g(s) + w h(s)



w = 2.5

13 expansions

11 movements

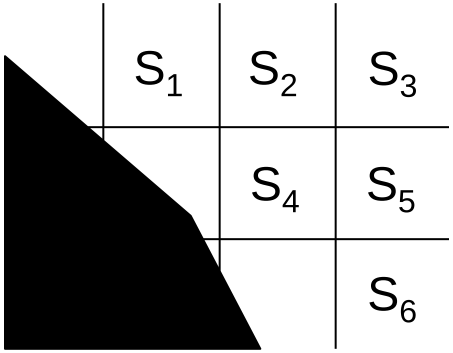
8-neighbor grid

Modeling Planning Domains

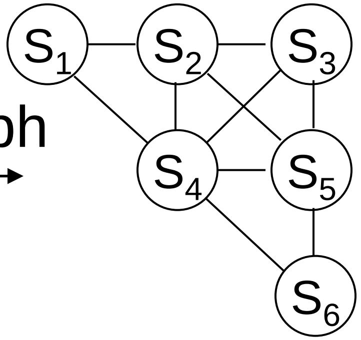
- Deterministic Models – Graphs
  - Skeletonization Methods (Roadmaps)
  - Cell Decomposition Methods
- Searching Graphs
  - A*
  - Weighted A*
- Nondeterministic Models – MDPs
- Searching MDPs

Modeling Uncertainty

- So far, we assumed no uncertainty in the model
  - execution is perfect
  - localization is perfect
  - environment is fully known



convert into a graph

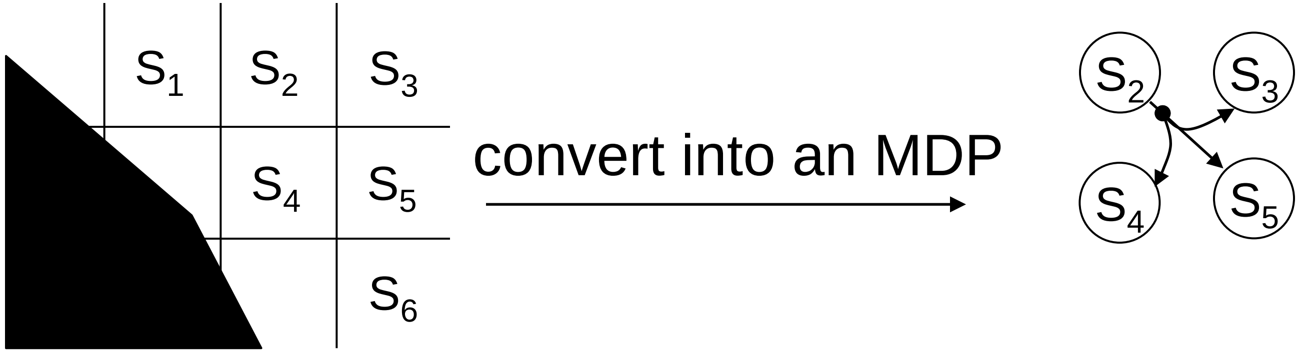


search the graph for a least-cost path from s_{start} to s_{goal}

Modeling Uncertainty

- Uncertainty in execution
    - **execution is imperfect**
    - localization is still assumed to be perfect
    - environment is still assumed to be fully known

Markov Decision Processes (MDP)

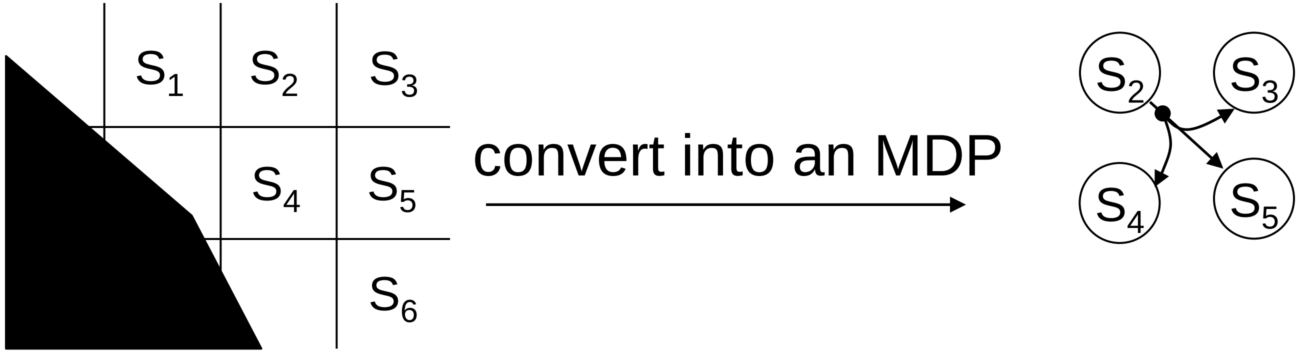


- at least one action in the graph has more than one outcome
- each outcome is associated with probability and cost

Modeling Uncertainty

- Uncertainty in execution
  - **execution is imperfect**
  - localization is still assumed to be perfect
  - environment is still assumed to be fully known

Markov Decision Processes (MDP)



- at least one action in the graph has more than one outcome
- each outcome is associated with probability and cost

example: $s_3, s_4, s_5 \in succ(s_2, a_{SE})$,

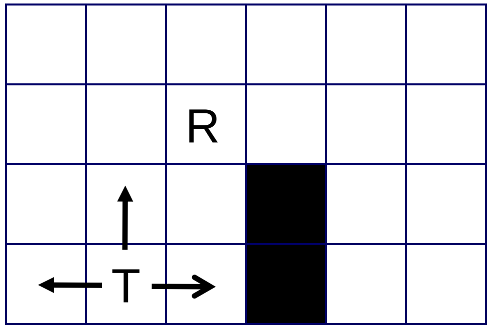
$$P(s_5|a_{se},s_2) = 0.9, \quad c(s_2,a_{se},s_5) = 1.4$$
$$P(s_3|a_{se},s_2) = 0.05, \quad c(s_2,a_{se},s_3) = 1.0$$
$$P(s_4|a_{se},s_2) = 0.05, \quad c(s_2,a_{se},s_4) = 1.0$$

Modeling Uncertainty

- Uncertainty in execution
  - **execution is imperfect**
  - localization is still assumed to be perfect
  - environment is still assumed to be fully known

Moving-target search example

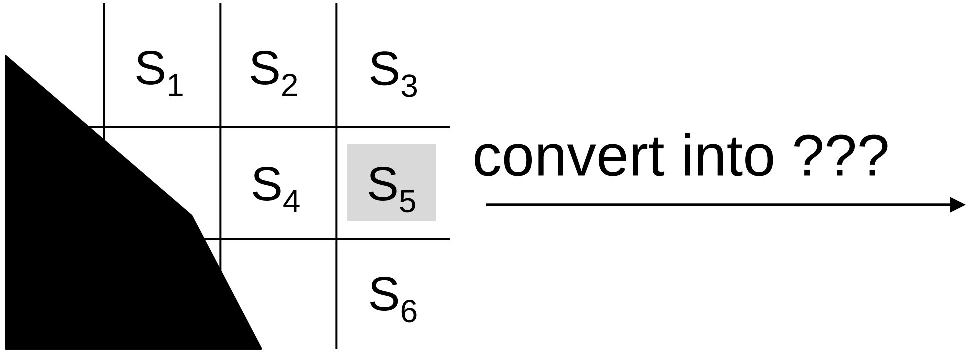
- State: *<R,T>*
- Uncertainty in the target moves



Modeling Uncertainty

- execution is perfect
- localization is still assumed to be perfect
- **environment is partially-known**

Incomplete Information State

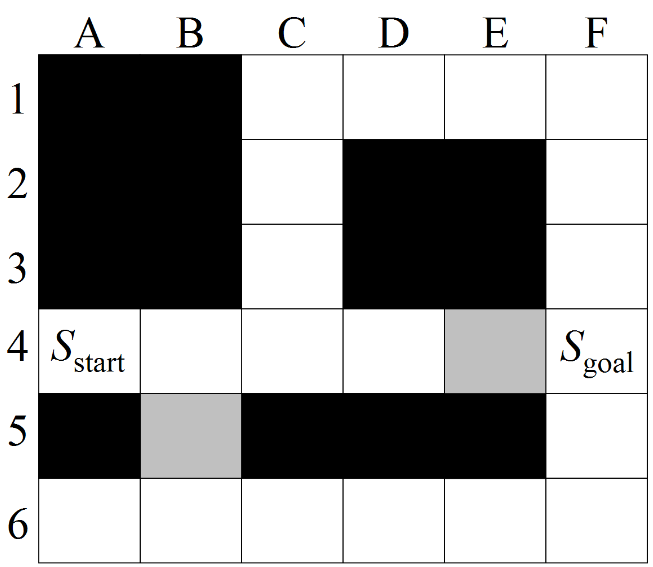


- the costs and connectivity of the graph is not fully known

Modeling Uncertainty

Information state (e.g., knowledge about the environment) is not fully known

Robot navigation in a partially-known environment



S – agent's state

<x,y> position

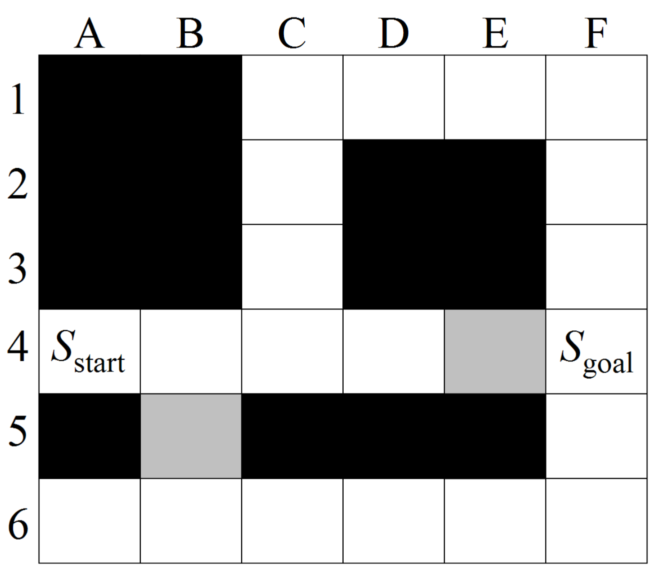
H – a vector of hidden variables

status of cells B5 and E4

Modeling Uncertainty

Information state (e.g., knowledge about the environment) is not fully known

Robot navigation in a partially-known environment



fully observable state variables (always known)

S – agent’s state

<x,y> position

H – a vector of hidden variables

status of cells B5 and E4

not known at the time of planning but probability distribution P(H) is given

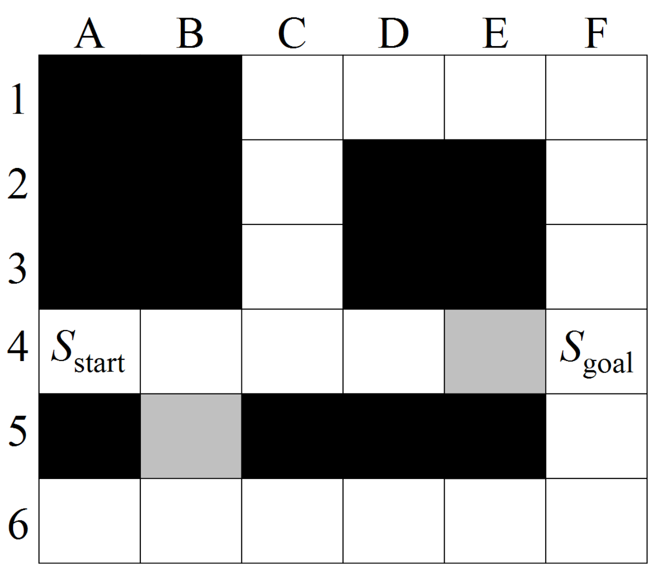
Modeling Uncertainty

X=[S(X);H(X)] - belief state

current (observable) state of the robot

current belief of the robot about hidden variables (i.e., P(H))

Robot navigation in a partially-known environment



S – agent's state

<x,y> position

H – a vector of hidden variables

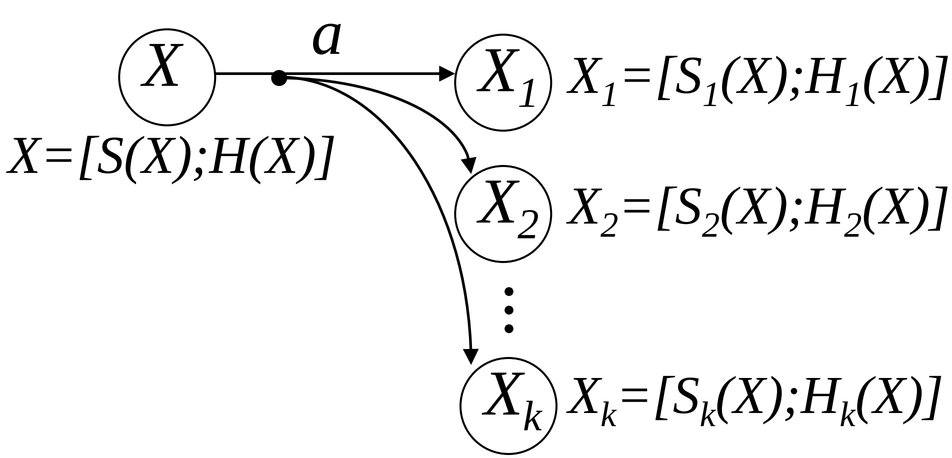
status of cells B5 and E4

Modeling Uncertainty: Incomplete Info State

Maxim

- Belief State-Space: *X=[S(X);H(X)] - belief state*

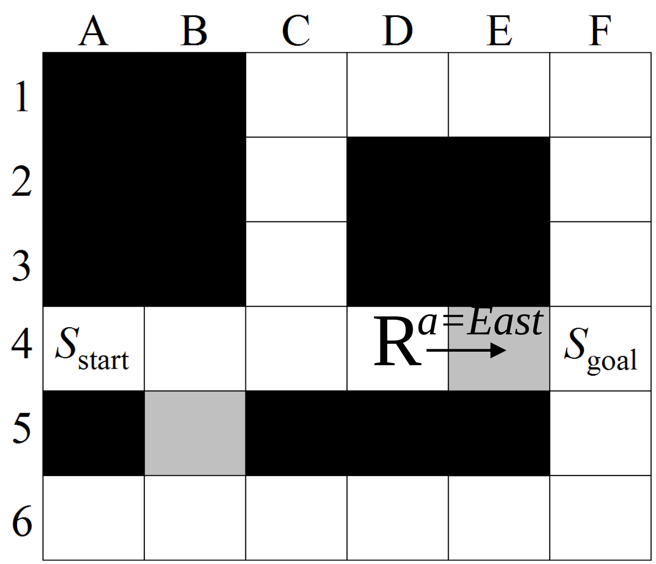
- An action can affect both the observable state of the robot (e.g., move action) as well as its knowledge about the environment (e.g., sensing action):

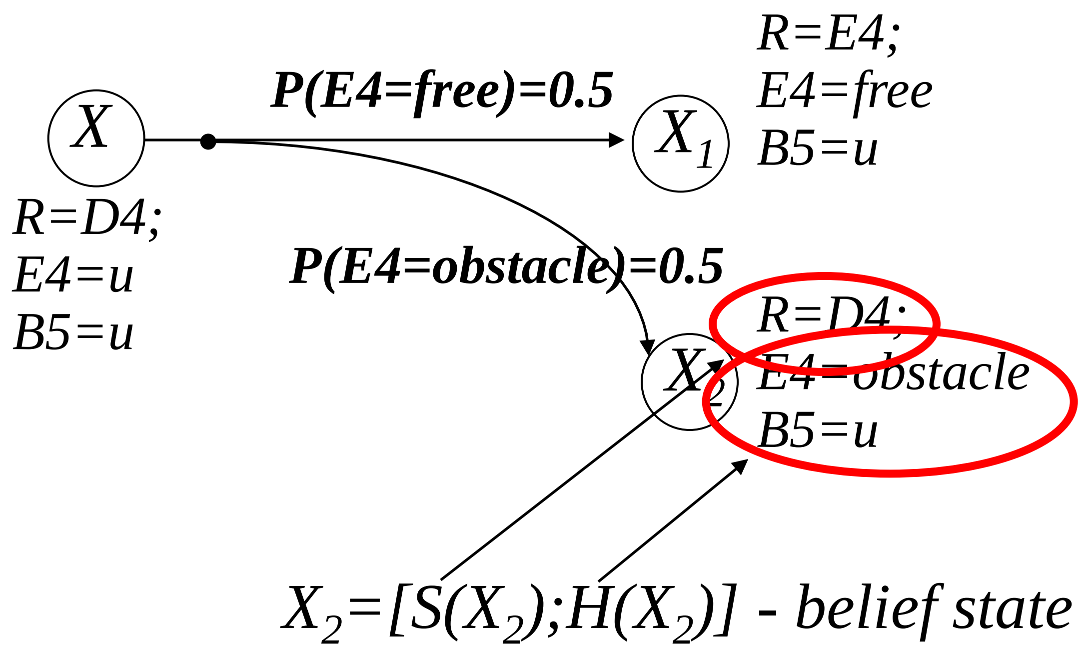


Modeling Uncertainty: Incomplete Info State

- Belief State-Space: *X=[S(X);H(X)] - belief state*

Assuming perfect sensing:



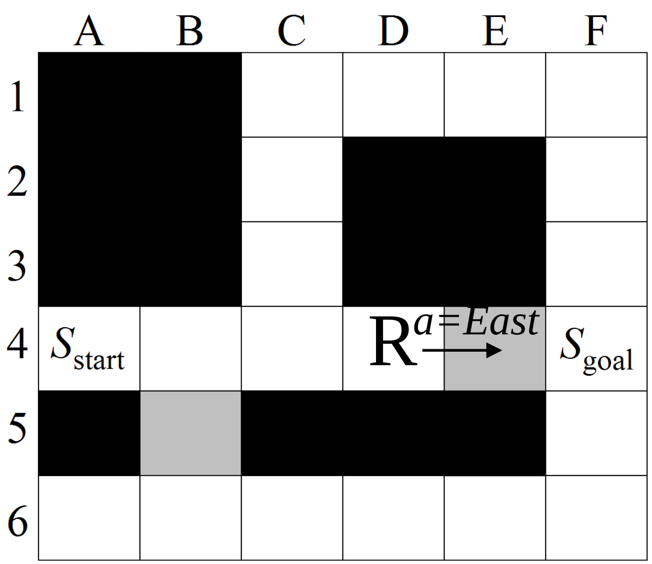


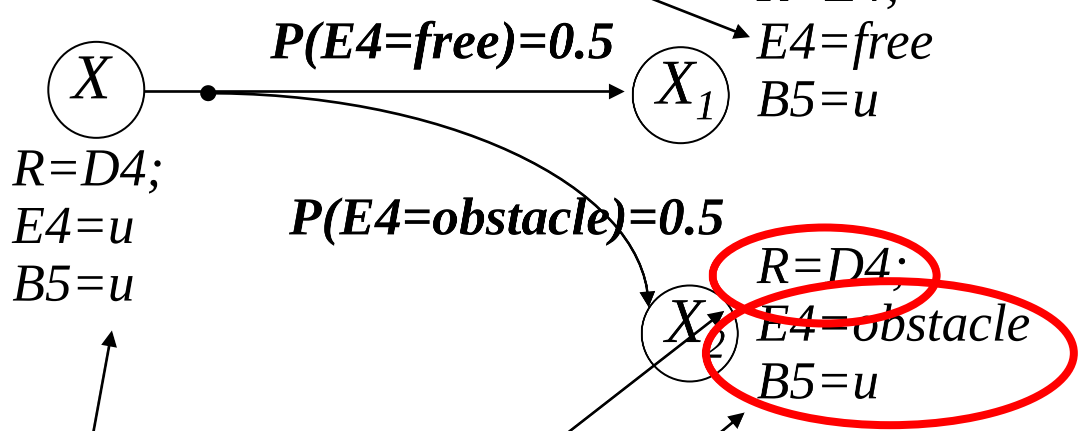
Modeling Uncertainty: Incomplete Info State

- Belief State-Space: *X=[S(X);H(X)] - belief state*

$H(X_1)$: P(E4=free) = 1; P(B5=free) = 0.5;

Assuming perfect sensing:





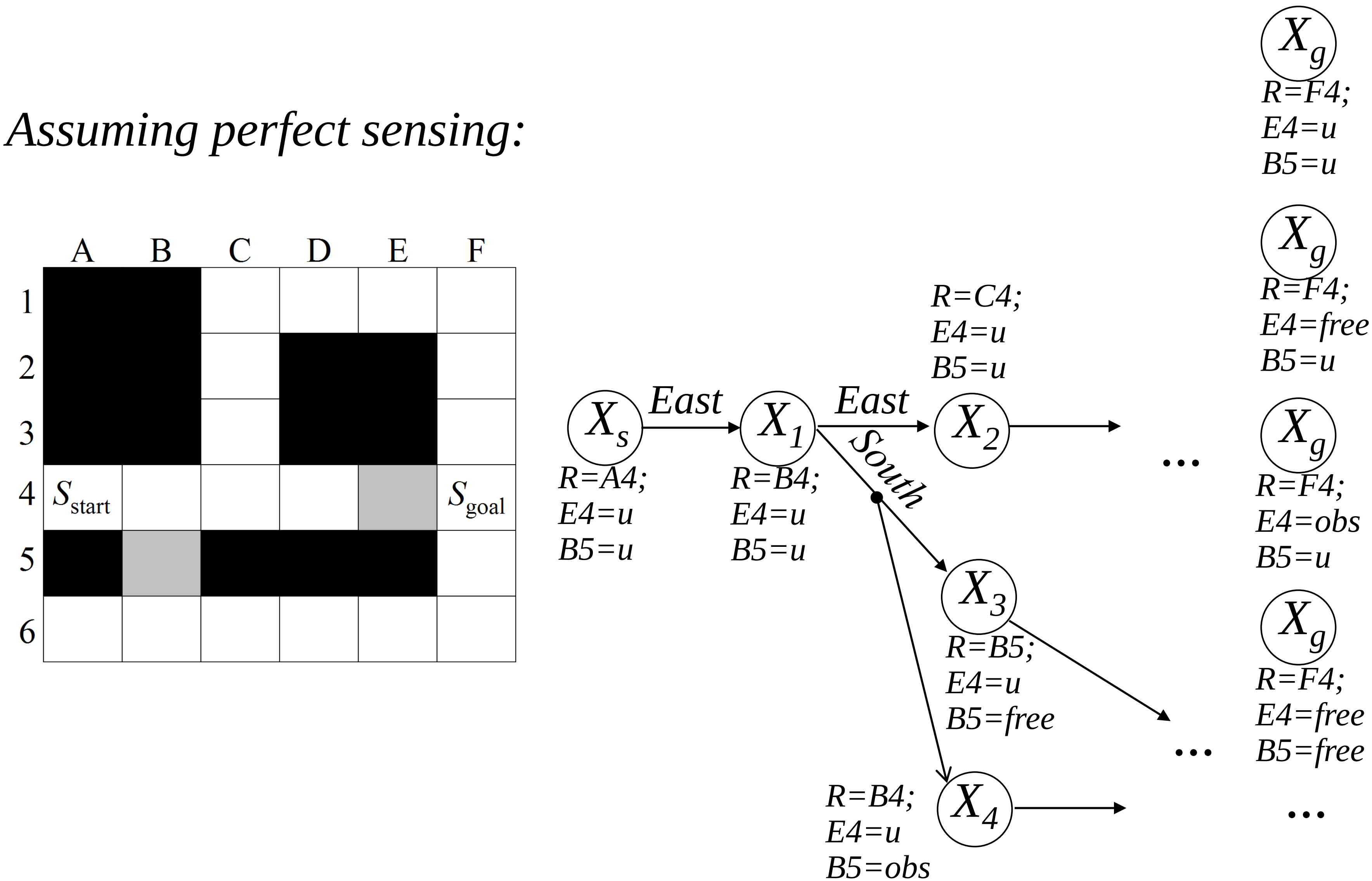
$X_2=[S(X_2);H(X_2)]$ - belief state

H(X): P(E4=free) = 0.5; P(B5=free) = 0.5;

Modeling Uncertainty: Incomplete Info State

- Belief State-Space: *X=[S(X);H(X)] - belief state*

Assuming perfect sensing:



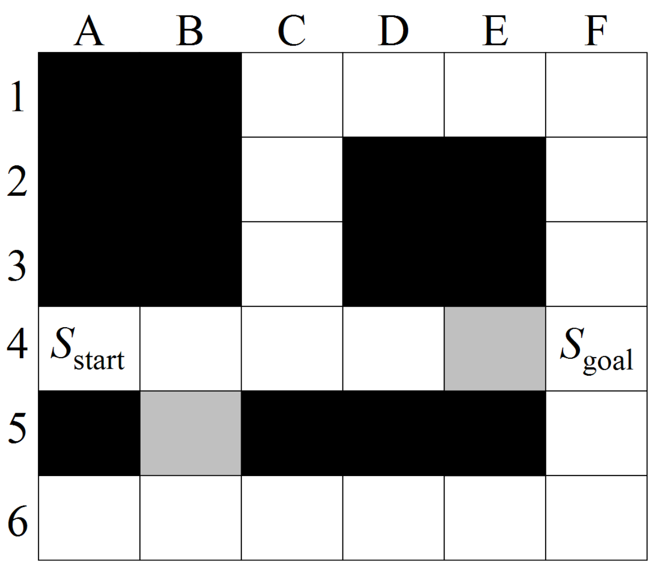
Modeling Uncertainty: Incomplete Info State

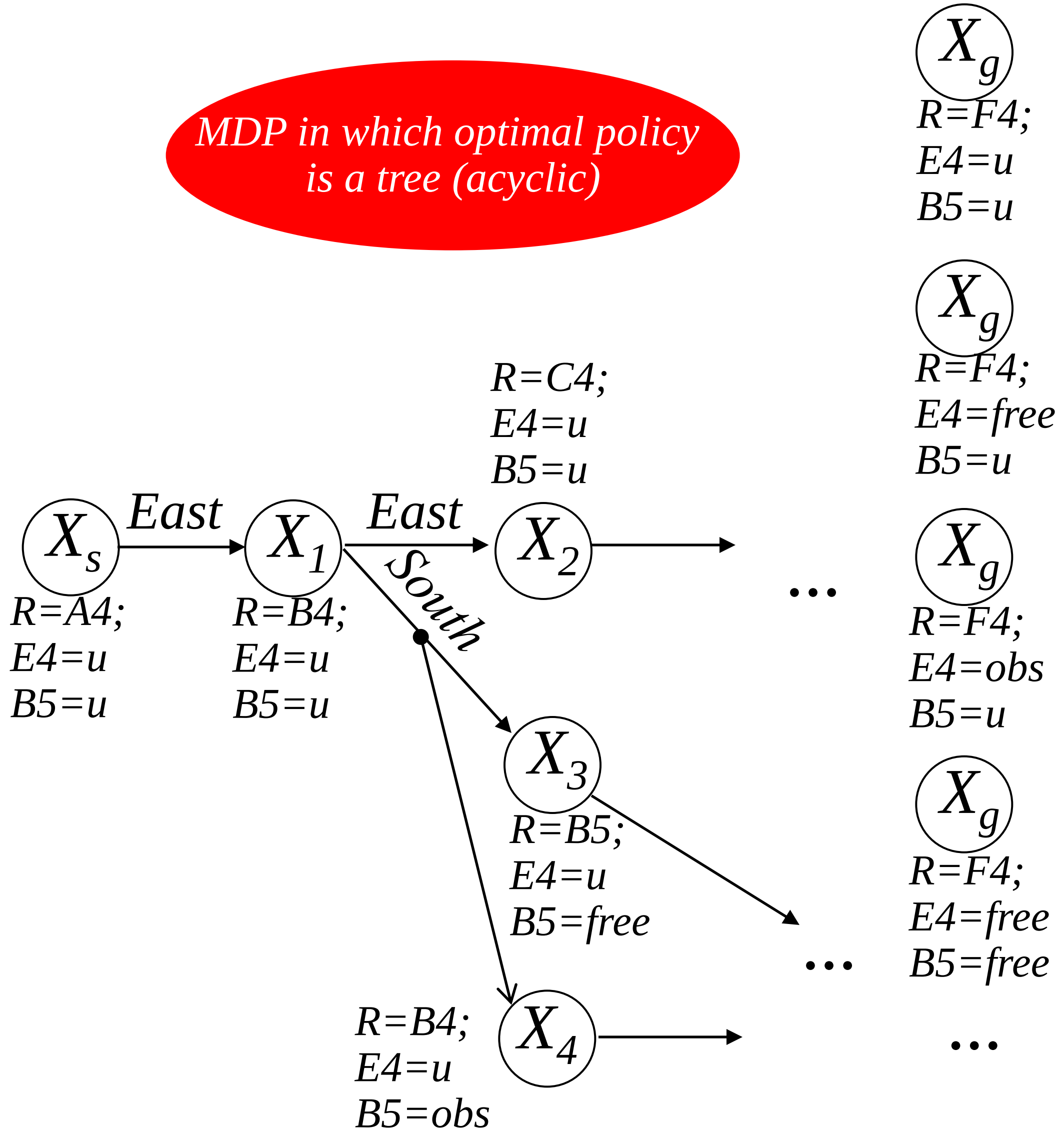
Maxim

- Belief State-Space: *X=[S(X);H(X)] - belief state*

Assuming perfect sensing:

MDP in which optimal policy is a tree (acyclic)





Modeling Uncertainty

- Uncertainty in localization/execution/environment
  - **execution is imperfect**
  - **localization is imperfect**
  - **environment is partially-known**

Partially-Observable MDPs (POMDPs)

Modeling Uncertainty: POMDPs

- MDP + robot is uncertain about its state (and/or about some of the action costs)

- Can always be converted into a belief state-space MDP

  (where each state is a probability distribution over original states)

- optimal policy: mapping from a belief state onto action

- optimal policy can now be **cyclic**

- optimal policy can be found by solving belief MDP

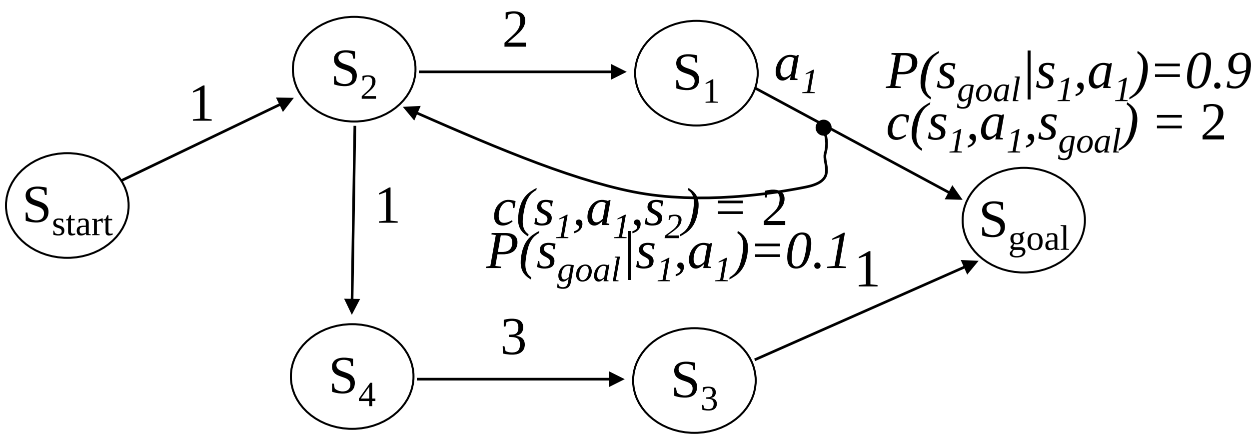
This tutorial will NOT talk about how to solve general POMDPs

Modeling Planning Domains

- Deterministic Models – Graphs
  - Skeletonization Methods (Roadmaps)
  - Cell Decomposition Methods
- Searching Graphs
  - A*
  - Weighted A*
- Nondeterministic Models – MDPs
- Searching MDPs

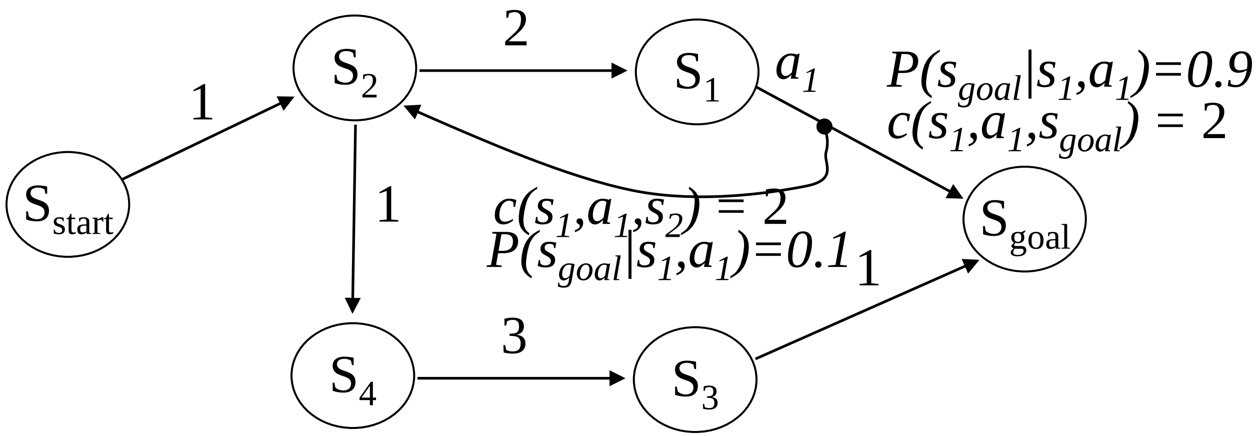
Probabilistic Planning

- What plan to compute?
  - Plan that minimizes the worst-case scenario (minimax plan)
  - Plan that minimizes the expected cost



- Without uncertainty, plan is a single path:
  a sequence of states (a sequence of actions)
- In MDPs, plan is a policy π:
  mapping from a state onto an action

Minimax Formulation

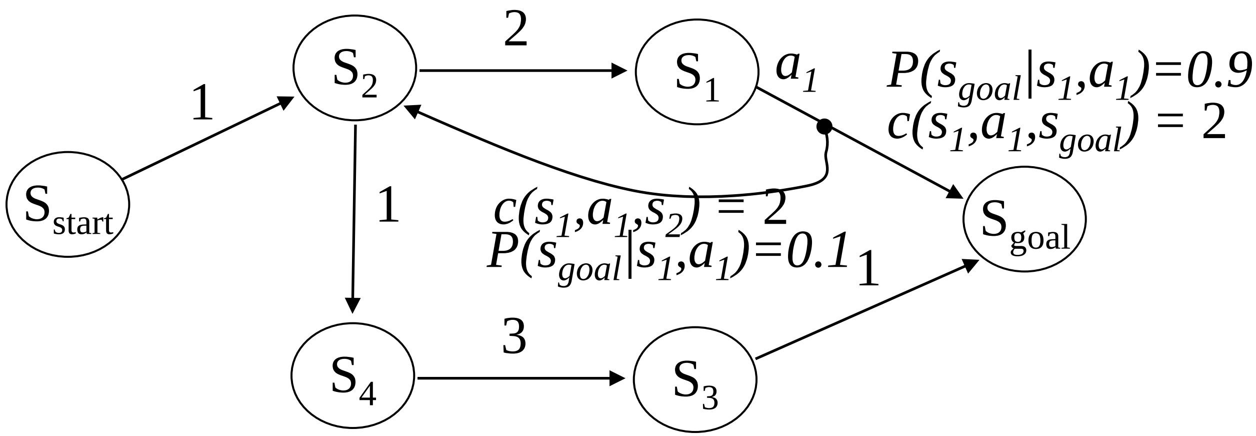


- Optimal policy π^*:
  minimizes the *worst* cost-to-goal
  $\pi^* = argmin_{\pi}\ max_{outcomes\ of\ \pi}\{cost\text{-}to\text{-}goal\}$

- worst cost-to-goal for $\pi_1=(s_{start},s_2,s_4,s_3,s_{goal})$ is:
  1+1+3+1 = 6

- worst cost-to-goal for π_2=(try to go through s_1) is:
  1+2+2+2+2+2+2 + … = ∞

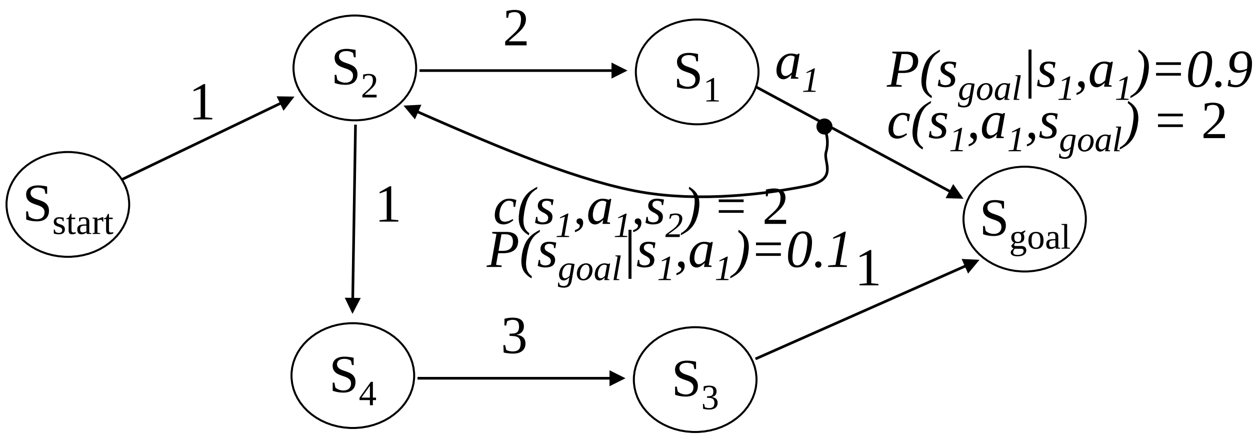
Minimax Formulation



- Optimal policy π^*:
  minimizes the *worst* cost-to-goal
  $\pi^* = argmin_{\pi}\ max_{outcomes\ of\ \pi}\{cost\text{-}to\text{-}goal\}$

- Optimal minimax policy $\pi^* = \pi_1 = (s_{start}, s_2, s_4, s_3, s_{goal})$

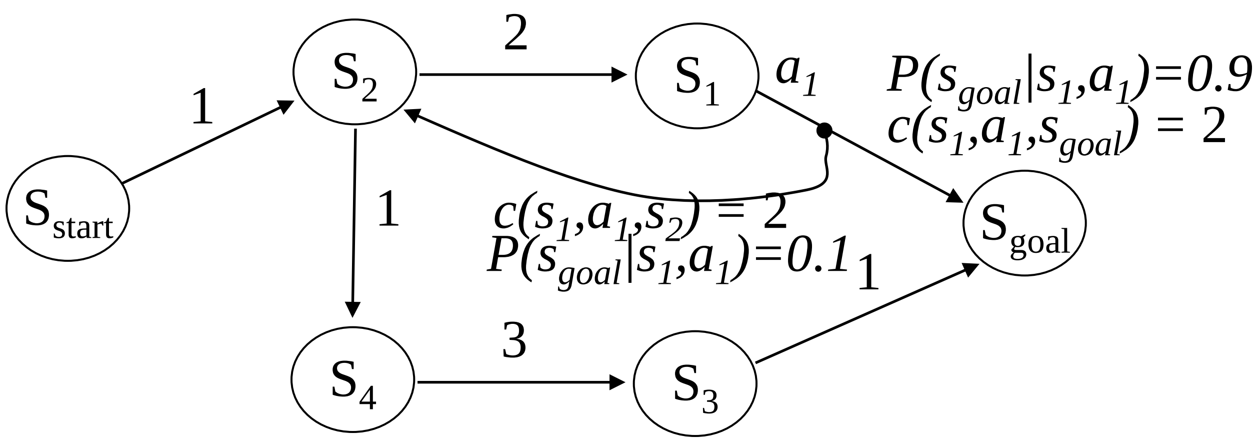
Computing Minimax Plans



- Minimax backward A*:

$g(s_{goal}) = 0$; all other g-values are infinite; $OPEN = \{s_{goal}\}$;
while(s_{start} *not expanded*)
  remove s with the smallest $[f(s) = g(s)+h(s)]$ from *OPEN*;
  insert s into *CLOSED*;
  for every s' s.t $s \in succ(s', a)$ for some a and s' not in *CLOSED*
    if $g(s') > \max_{u \in succ(s', a)} c(s',u) + g(u)$
      $g(s') = \max_{u \in succ(s', a)} c(s',u) + g(u)$;
      insert s' into *OPEN*;

Computing Minimax Plans

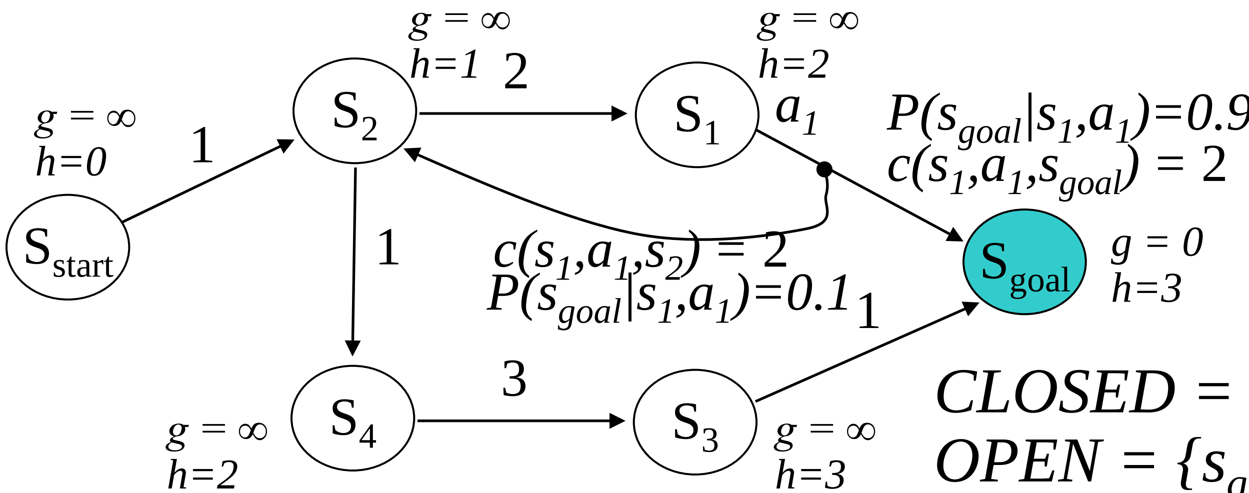


- Minimax backward A*:

$g(s_{goal}) = 0;$ all other g-values are infinite; $OPEN = \{s_{goal}\};$
while(s_{start} *not expanded*)
  remove s with the smallest $[f(s) = g(s)+h(s)]$ from *OPEN*;
  insert s into *CLOSED*;
  for every s' s.t $s \in succ(s', a)$ for some a and s' not in *CLOSED*
    if $g(s') > max_{u \in succ(s', a)}\, c(s',u) + g(u)$
      $g(s') = max_{u \in succ(s', a)}\, c(s',u) + g(u);$
      insert s' into *OPEN*;

reduces to usual backward A if no uncertainty in outcomes*

Computing Minimax Plans

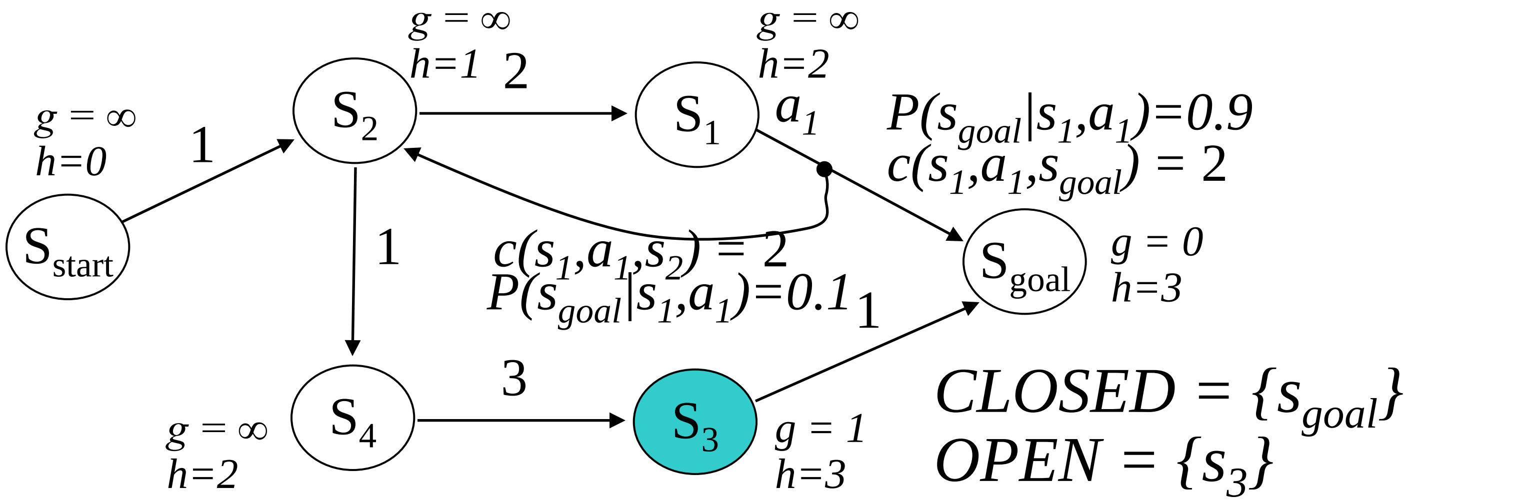


CLOSED = {}
OPEN = {s_{goal}}
next state to expand: s_{goal}

- Minimax backward A*:

$g(s_{goal}) = 0$; all other g-values are infinite; $OPEN = \{s_{goal}\}$;
while(s_{start} *not expanded*)
  remove s with the smallest $[f(s) = g(s)+h(s)]$ from *OPEN*;
  insert s into *CLOSED*;
  for every s' s.t $s \in succ(s', a)$ for some a and s' not in *CLOSED*
    if $g(s') > \max_{u \in succ(s', a)} c(s',u) + g(u)$
      $g(s') = \max_{u \in succ(s', a)} c(s',u) + g(u)$;
      insert s' into *OPEN*;

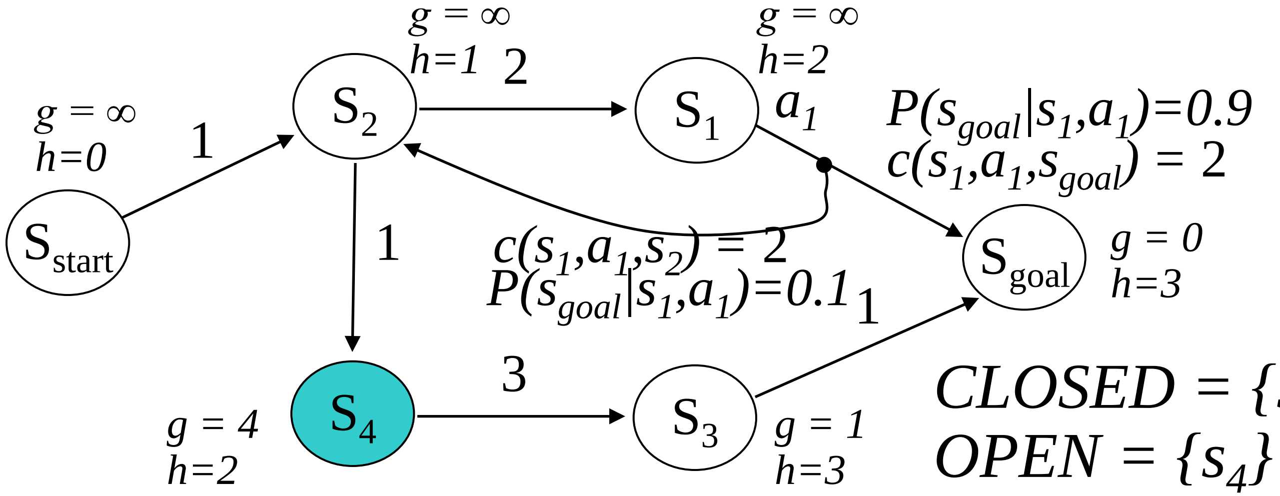
Computing Minimax Plans



- Minimax backward A*:

$g(s_{goal}) = 0$; all other g-values are infinite; $OPEN = \{s_{goal}\}$;
while(s_{start} *not expanded*)
  remove s with the smallest $[f(s) = g(s)+h(s)]$ from *OPEN*;
  insert s into *CLOSED*;
  for every s' s.t $s \in succ(s', a)$ for some a and s' not in *CLOSED*
    if $g(s') > max_{u \in succ(s', a)} c(s',u) + g(u)$
      $g(s') = max_{u \in succ(s', a)} c(s',u) + g(u)$;
      insert s' into *OPEN*;

Computing Minimax Plans

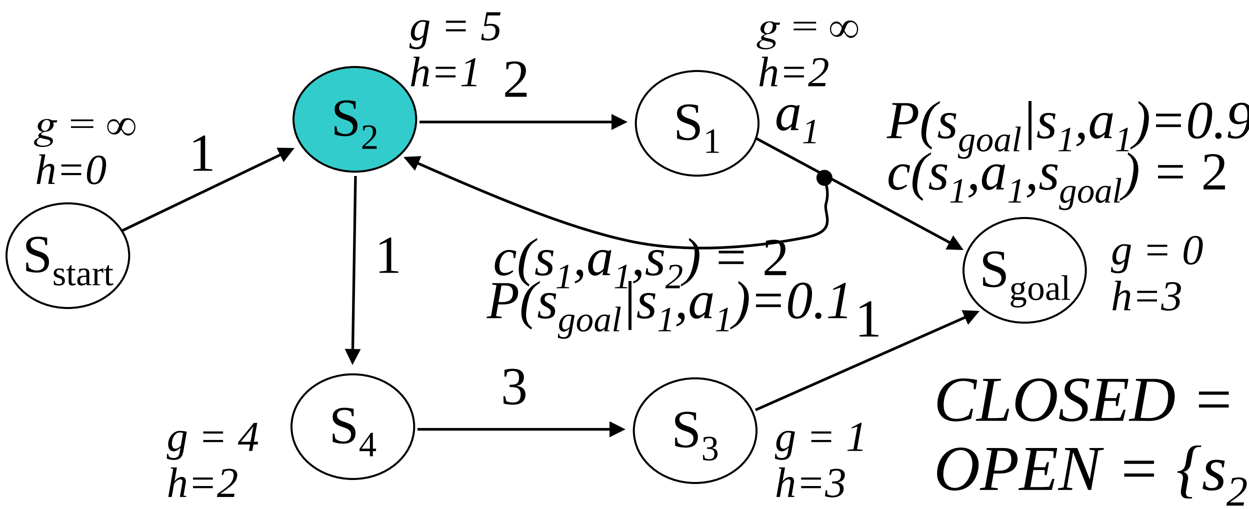


$CLOSED = \{s_{goal}, s_3\}$
$OPEN = \{s_4\}$
next state to expand: s_4

- Minimax backward A*:

$g(s_{goal}) = 0$; all other g-values are infinite; $OPEN = \{s_{goal}\}$;
while(s_{start} *not expanded*)
  remove s with the smallest $[f(s) = g(s)+h(s)]$ from $OPEN$;
  insert s into $CLOSED$;
  for every s' s.t $s \in succ(s', a)$ for some a and s' not in $CLOSED$
    if $g(s') > \max_{u \in succ(s', a)} c(s',u) + g(u)$
      $g(s') = \max_{u \in succ(s', a)} c(s',u) + g(u)$;
      insert s' into $OPEN$;

Computing Minimax Plans

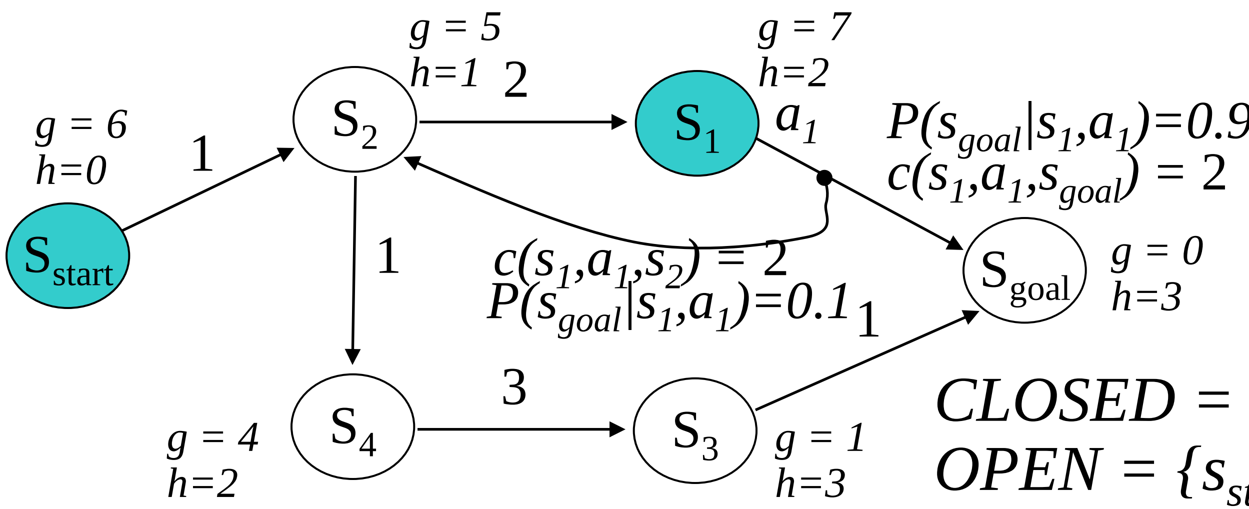


CLOSED = $\{s_{goal}, s_3, s_4\}$
OPEN = $\{s_2\}$
next state to expand: s_2

- Minimax backward A*:

$g(s_{goal}) = 0$; all other g-values are infinite; *OPEN* = $\{s_{goal}\}$;
while(s_{start} *not expanded*)
  remove s with the smallest $[f(s) = g(s)+h(s)]$ from *OPEN*;
  insert s into *CLOSED*;
  for every s' s.t $s \in succ(s', a)$ for some a and s' not in *CLOSED*
    if $g(s') > \max_{u \in succ(s', a)} c(s',u) + g(u)$
      $g(s') = \max_{u \in succ(s', a)} c(s',u) + g(u)$;
      insert s' into *OPEN*;

Computing Minimax Plans

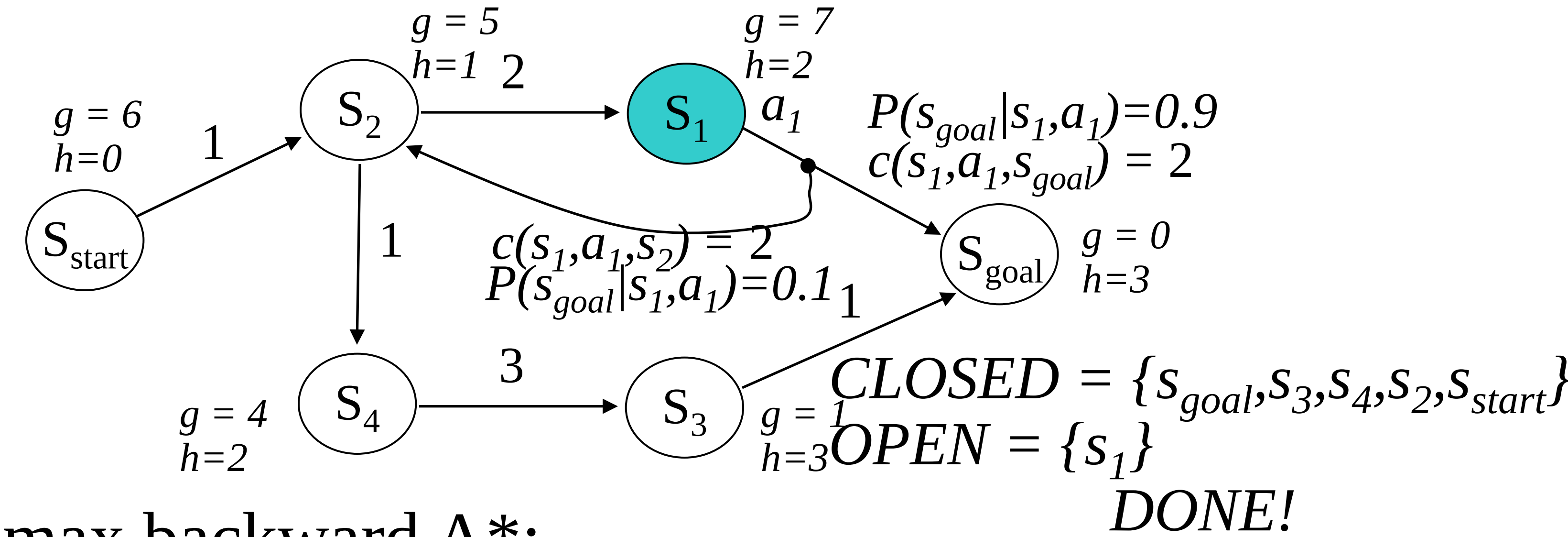


$CLOSED = \{s_{goal},s_3,s_4,s_2\}$
$OPEN = \{s_{start},s_1\}$
next state to expand: s_{start}

- Minimax backward A*:

$g(s_{goal}) = 0;$ all other g-values are infinite; $OPEN = \{s_{goal}\};$
while(s_{start} *not expanded)*
  remove s with the smallest $[f(s) = g(s)+h(s)]$ from *OPEN*;
  insert s into *CLOSED*;
  for every s' s.t $s \in succ(s', a)$ for some a and s' not in *CLOSED*
    if $g(s') > \max_{u \in succ(s', a)} c(s',u) + g(u)$
      $g(s') = \max_{u \in succ(s', a)} c(s',u) + g(u);$
      insert s' into *OPEN*;

Computing Minimax Plans



- Minimax backward A*:

$g(s_{goal}) = 0$; all other g-values are infinite; $OPEN = \{s_{goal}\}$;

while(s_{start} *not expanded*)

remove s with the smallest $[f(s) = g(s)+h(s)]$ from $OPEN$;

insert s into $CLOSED$;

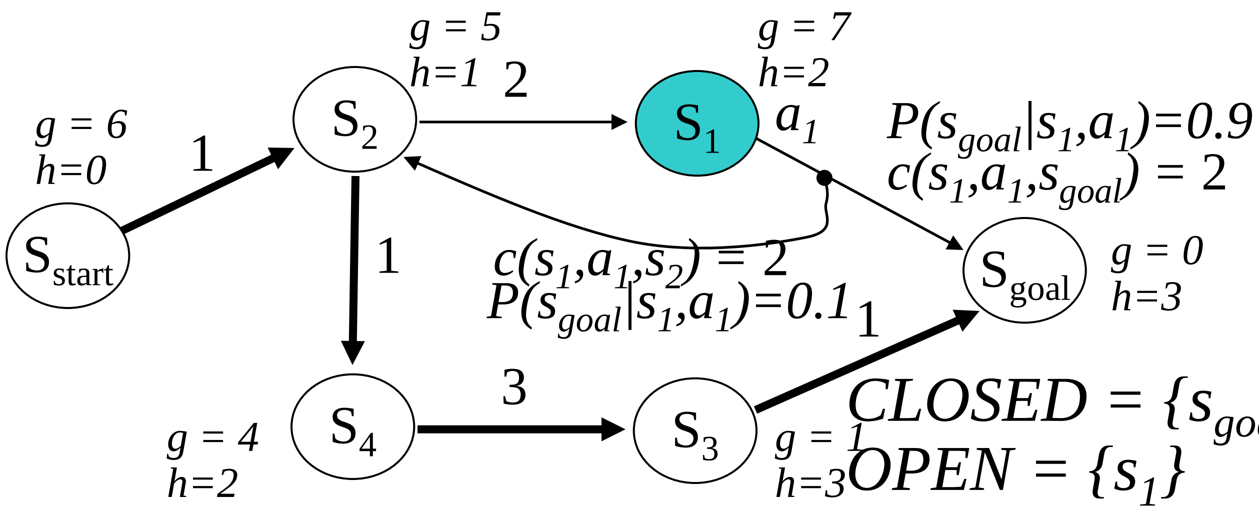
for every s' s.t $s \in succ(s', a)$ for some a and s' not in $CLOSED$

if $g(s') > \max_{u \in succ(s', a)} c(s',u) + g(u)$

$g(s') = \max_{u \in succ(s', a)} c(s',u) + g(u)$;

insert s' into $OPEN$;

Computing Minimax Plans



$CLOSED = \{s_{goal},s_3,s_4,s_2,s_{start}\}$

$OPEN = \{s_1\}$

DONE!

- Minimax backward A*:

$g(s_{goal}) = 0$; all other g-values are infinite; $OPEN = \{s_{goal}\}$;

while(s_{start} *not expanded*)

remove s with the smallest $[f(s) = g(s)+h(s)]$ from $OPEN$;

insert s into $CLOSED$;

for every s' s.t $s \in succ(s', a)$ for some a and s' not in $CLOSED$

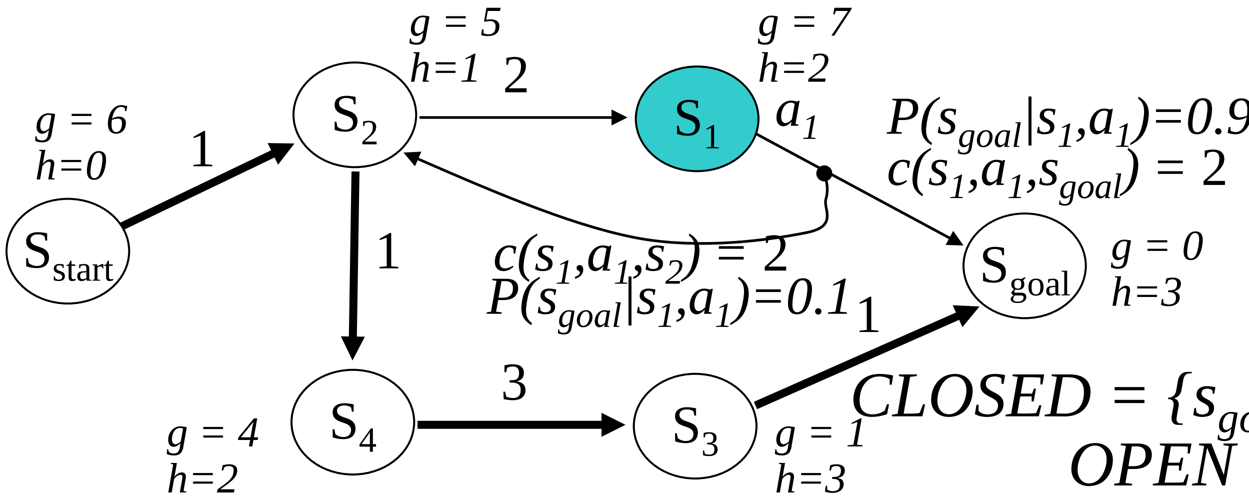
if $g(s') > max_{u \in succ(s', a)}\ c(s',u) + g(u)$

$g(s') = max_{u \in succ(s', a)}\ c(s',u) + g(u)$;

insert s' into $OPEN$;

in this example, the computed policy is a path, but in general it is a tree

Computing Minimax Plans



$CLOSED = \{s_{goal}, s_3, s_4, s_2, s_{start}\}$

$OPEN = \{s_1\}$

DONE!

- Minimax backward A*:

$g(s_{goal}) = 0$; all other g-values are infinite; $OPEN = \{s_{goal}\}$;

while(s_{start} *not expanded*)

remove s with the smallest $[f(s) = g(s)+h(s)]$ from $OPEN$;

insert s into $CLOSED$;

for every s' s.t $s \in succ(s', a)$ for some a and s' not in $CLOSED$

if $g(s') > max_{u \in succ(s', a)}\, c(s',u) + g(u)$

$g(s') = max_{u \in succ(s', a)}\, c(s',u)$

insert s' into $OPEN$;

Minimax A guarantees*
to find an optimal (minimax) policy,
and never expands a state more than once,
provided heuristics are consistent (just like A)*

Computing Minimax Plans

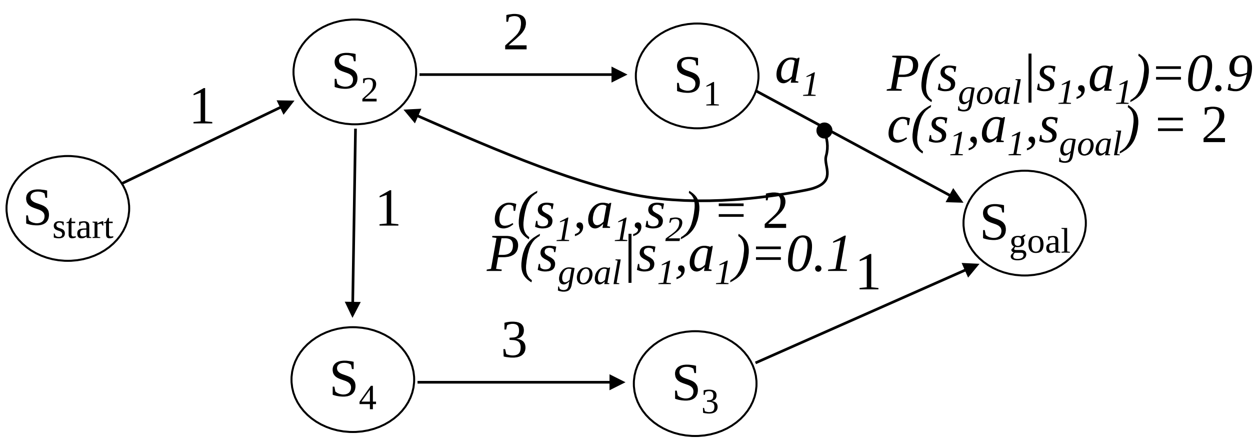
- Minimax backward A*
  - searches backwards which sometimes can be hard/computationally very expensive (consider moving-target search, what is a goal?)

Computing Minimax Plans

- Pros/cons of minimax plans
  - robust to uncertainty
  - overly pessimistic
  - harder to compute than normal paths
    - especially if backwards minimax A* does not apply
    - even if backwards minimax A* does apply, still more expensive than computing a single path with A* (heuristics are not guiding well)

Expected Cost Formulation

Maxim



- Optimal policy π^*:
  minimizes the *expected* cost-to-goal
  $\pi^* = argmin_\pi \, E\{cost\text{-}to\text{-}goal\}$

expectation over outcomes

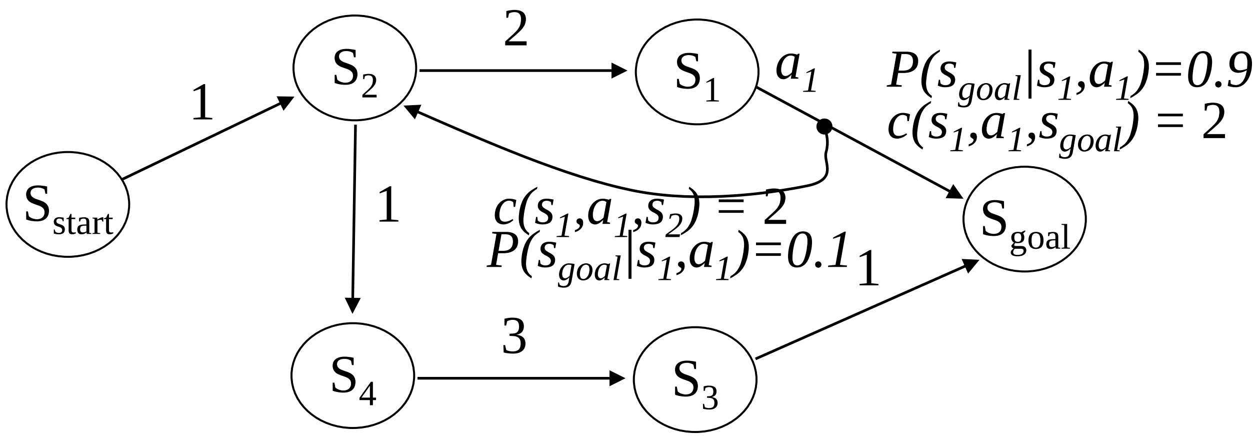
- expected cost-to-goal for $\pi_1=(s_{start},s_2,s_4,s_3,s_{goal})$ is

  1+1+3+1=6

- cost-to-goal for π_2=(try to go through s_1) is:

0.9*(1+2+2) + 0.9*0.1*(1+2+2+2+2) + 0.9*0.1*0.1*(1+2+2+2+2+2+2) + …=$5.44\overline{4}$

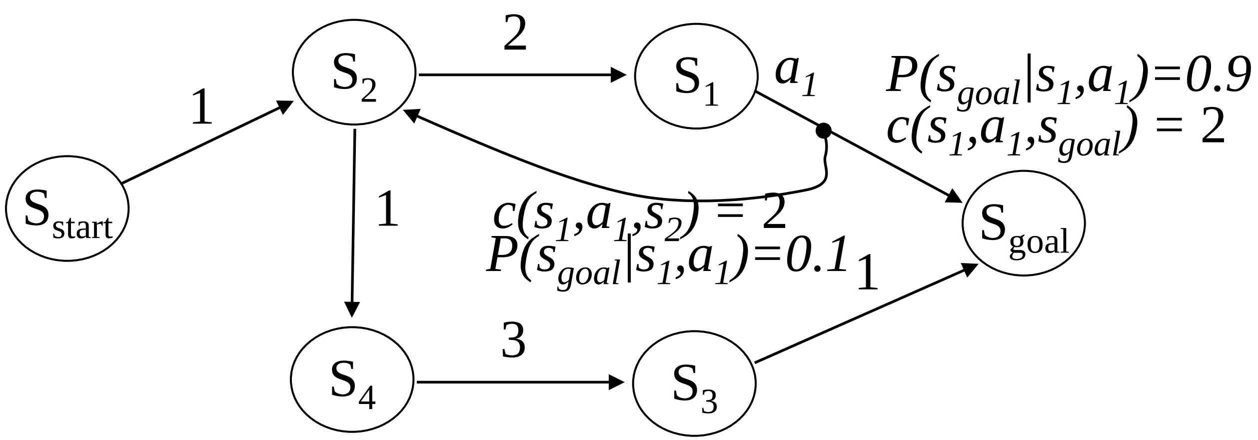
Expected Cost Formulation



- Optimal policy π^*:
  minimizes the *expected* cost-to-goal
  $\pi^* = argmin_\pi \, E\{cost\text{-}to\text{-}goal\}$

- Optimal expected cost policy $\pi^* = \pi_2 = (go\ through\ s_1)$

Computing Expected Cost Minimal Plans



- Optimal expected cost-to-goal values v^* satisfy:

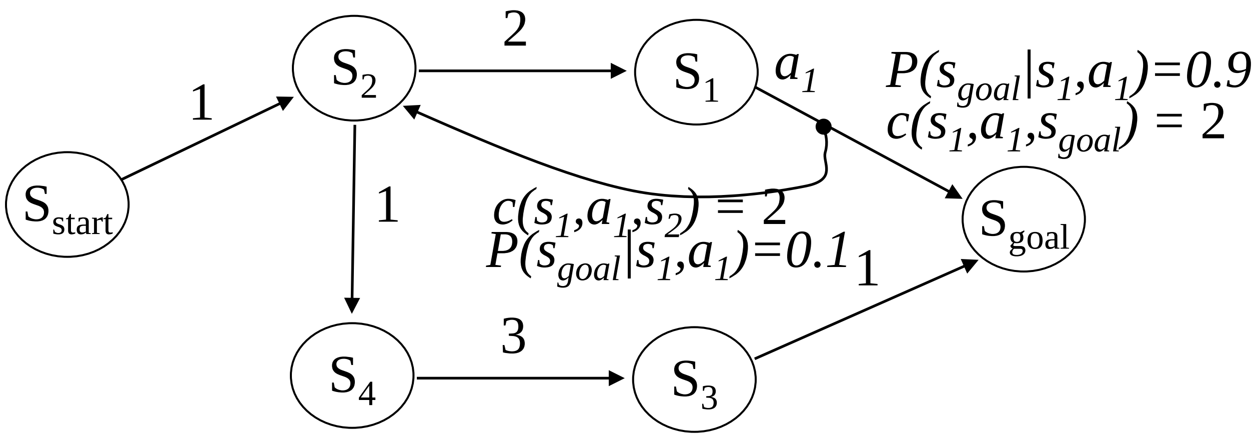
  $v^*(s_{goal})=0$

  $v^*(s) = \min_a E\{c(s,a,s')+v^*(s')\}$ *for all* $s \neq s_{goal}$

  (expectation over outcomes s' of action a executed at state s)

Bellman optimality equation

Computing Expected Cost Minimal Plans



- Value Iteration (VI):

  Initialize *v*-values of all states to finite values;

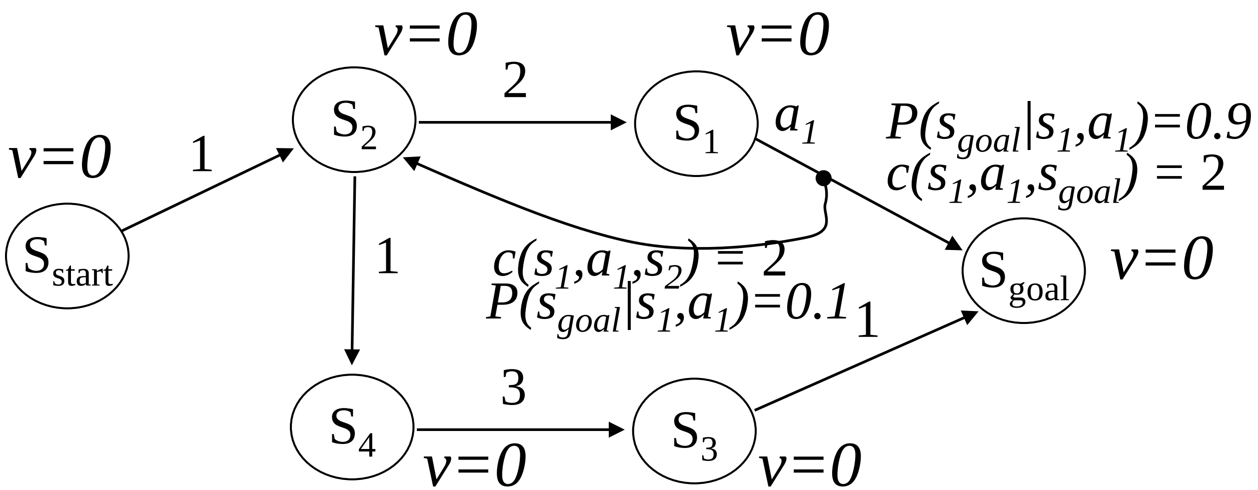
  Iterate over all *s* in MDP and re-compute until convergence:

  $v(s_{goal}) = 0$

  $v(s) = \min_a E\{c(s,a,s')+v(s')\}$ *for any* $s \neq s_{goal}$

Bellman update equation (or backup)

Computing Expected Cost Minimal Plans



- Value Iteration (VI):

  Initialize *v*-values of all states to finite values;

  Iterate over all *s* in MDP and re-compute until convergence:

  $v(s_{goal}) = 0$

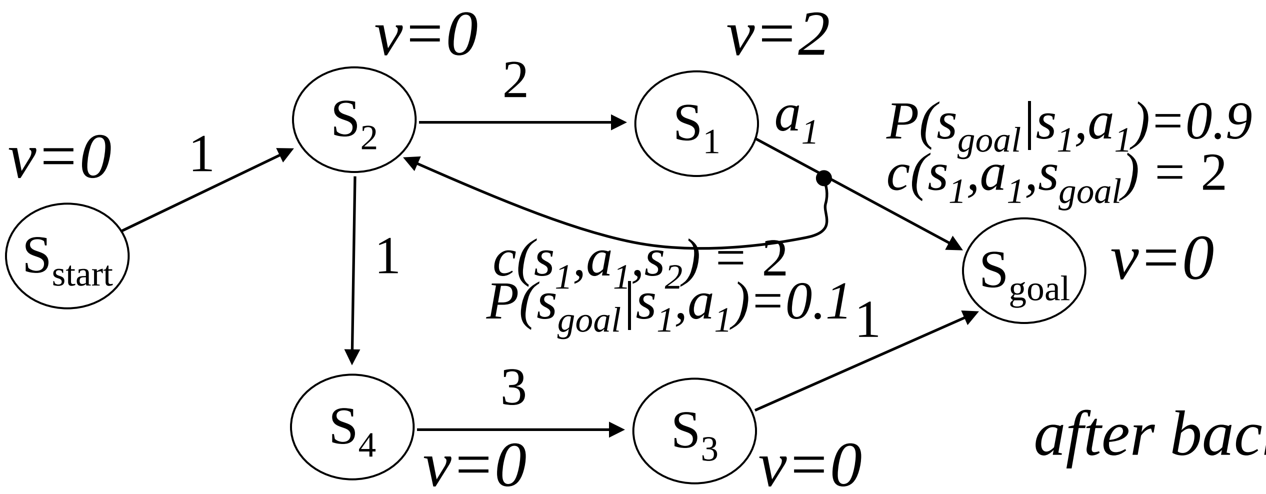
  $v(s) = \min_a E\{c(s,a,s') + v(s')\}$ *for any* $s \neq s_{goal}$

↑

Bellman update equation
(or backup)

Computing Expected Cost Minimal Plans

Maxim



after backing up s_1

- Value Iteration (VI):

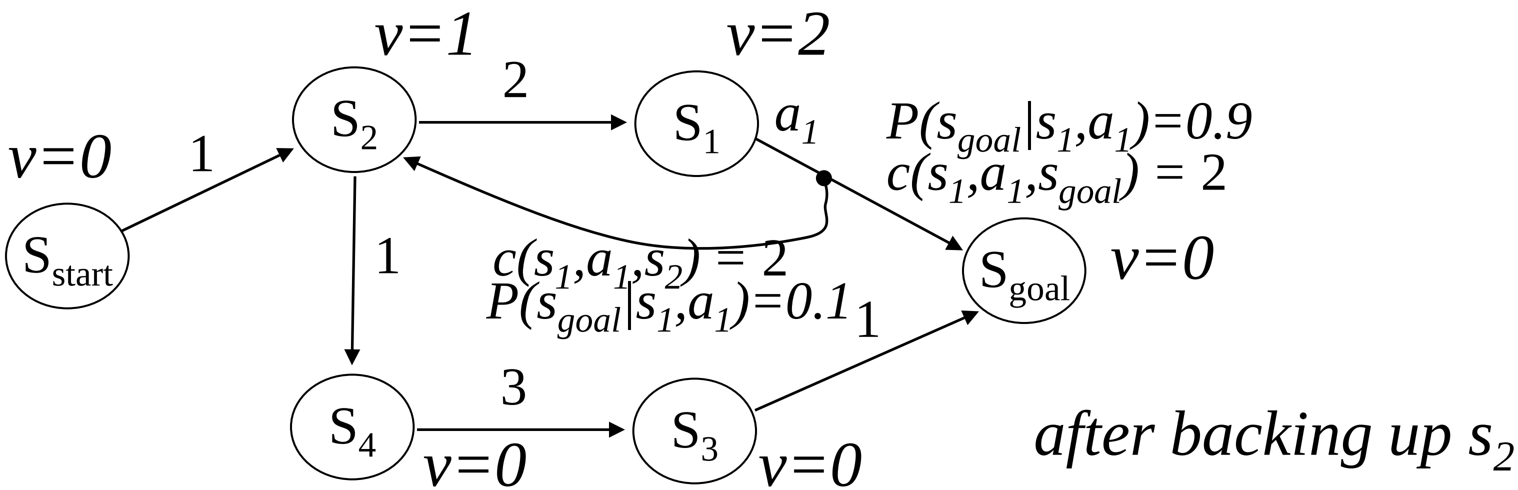
  Initialize *v*-values of all states to finite values;

  Iterate over all *s* in MDP and re-compute until convergence:

  $v(s_{goal}) = 0$

  $v(s) = \min_a E\{c(s,a,s')+v(s')\}$ *for any* $s \neq s_{goal}$

Computing Expected Cost Minimal Plans



- Value Iteration (VI):

  Initialize *v*-values of all states to finite values;

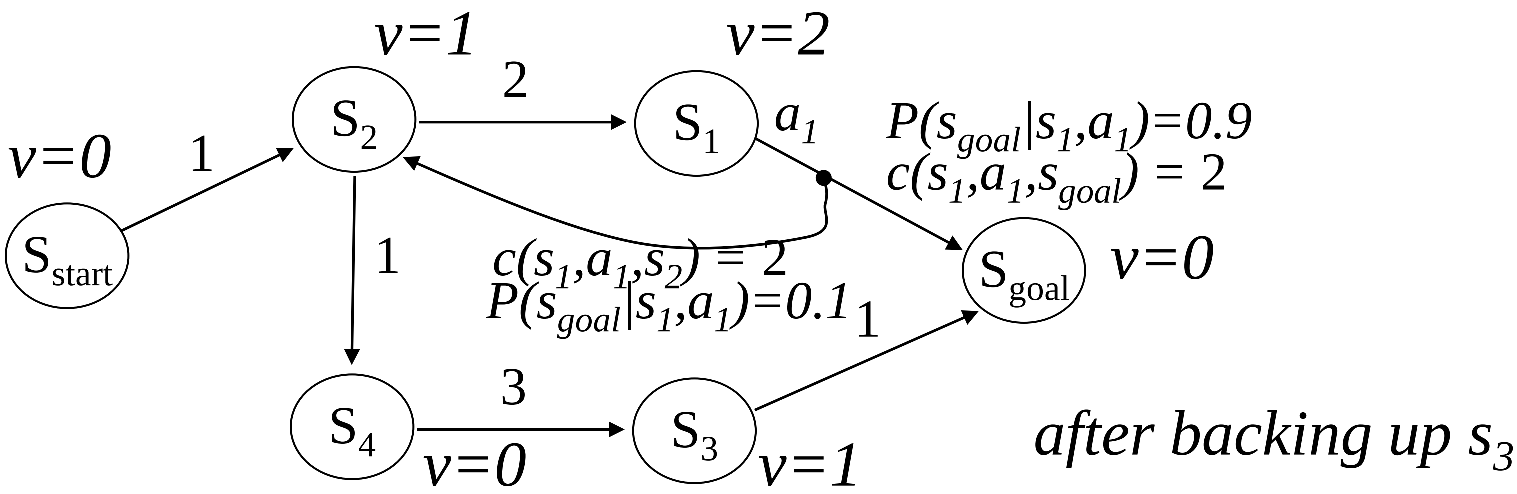
  Iterate over all *s* in MDP and re-compute until convergence:

  $v(s_{goal}) = 0$

  $v(s) = \min_a E\{c(s,a,s')+v(s')\}$ *for any* $s \neq s_{goal}$

Computing Expected Cost Minimal Plans

Maxim



- Value Iteration (VI):

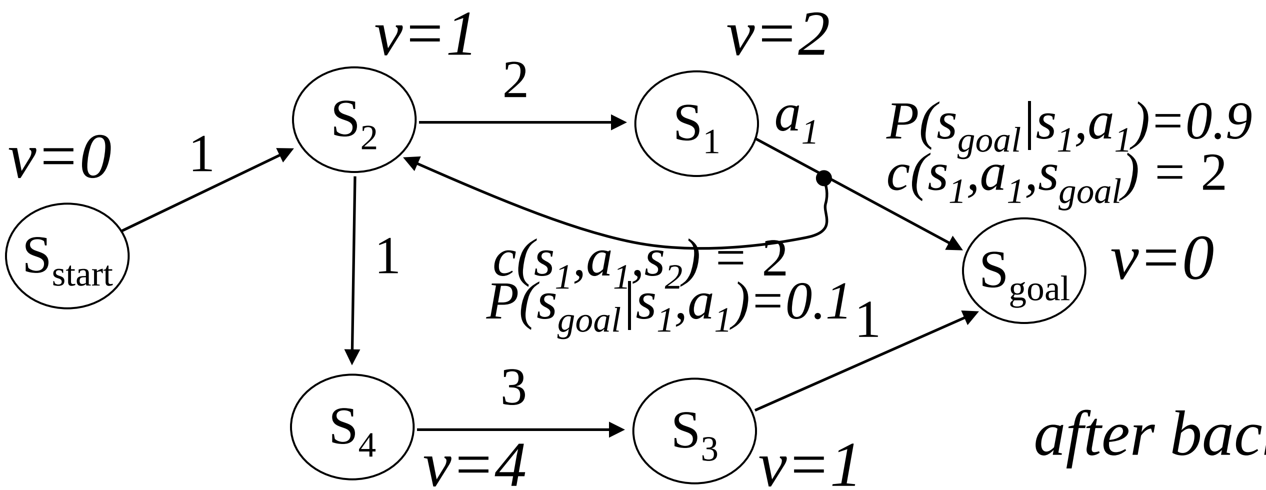
  Initialize *v*-values of all states to finite values;

  Iterate over all *s* in MDP and re-compute until convergence:

  $v(s_{goal}) = 0$

  $v(s) = \min_a E\{c(s,a,s')+v(s')\}$ for any $s \neq s_{goal}$

Computing Expected Cost Minimal Plans



after backing up s_4

- Value Iteration (VI):

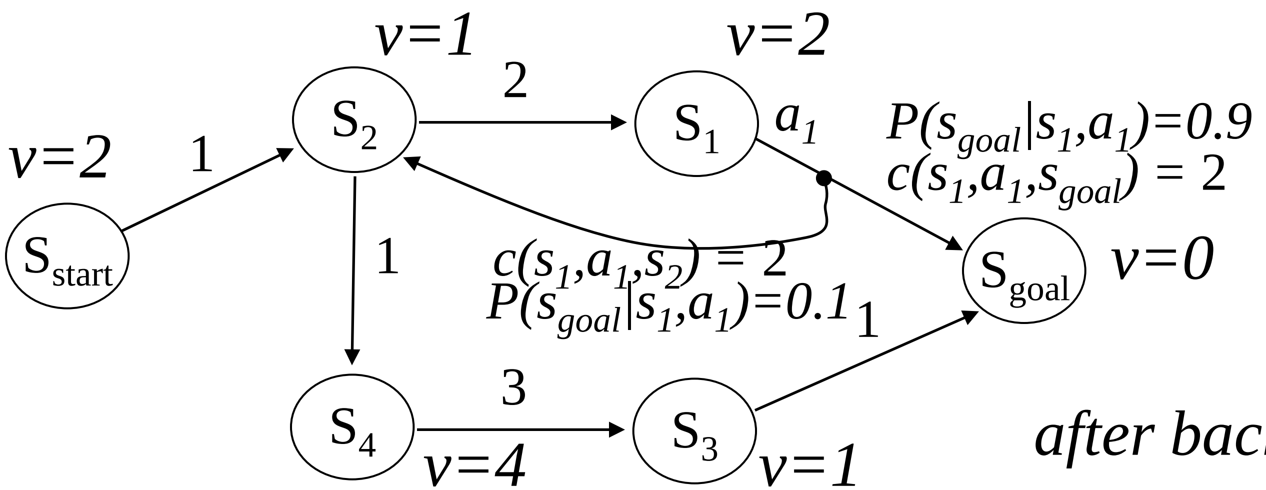
  Initialize *v*-values of all states to finite values;

  Iterate over all *s* in MDP and re-compute until convergence:

  $v(s_{goal}) = 0$

  $v(s) = \min_a E\{c(s,a,s')+v(s')\}$ *for any* $s \neq s_{goal}$

Computing Expected Cost Minimal Plans



after backing up s_{start}

- Value Iteration (VI):

  Initialize *v*-values of all states to finite values;

  Iterate over all *s* in MDP and re-compute until convergence:

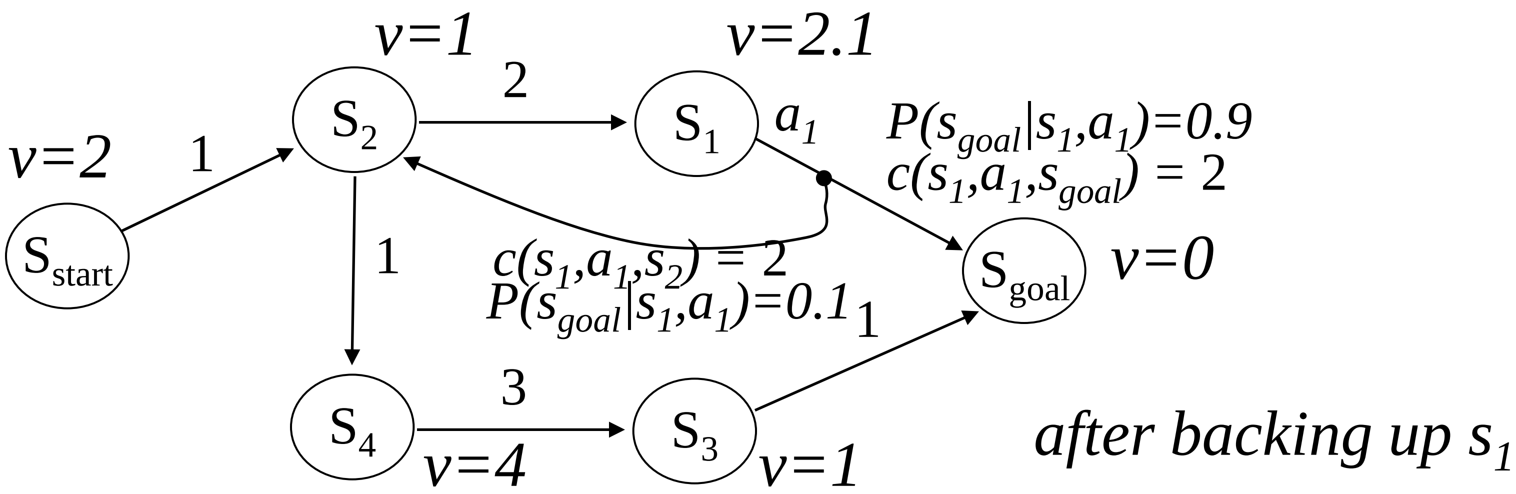
  $v(s_{goal}) = 0$

  $v(s) = \min_a E\{c(s,a,s') + v(s')\}$ *for any* $s \neq s_{goal}$

Usual convergence condition: Bellman error over all states $< \Delta$

Bellman error: $|v(s) - \min_a E\{c(s,a,s') + v(s')\}|$ *for any* $s \neq s_{goal}$

Computing Expected Cost Minimal Plans



- Value Iteration (VI):

  Initialize *v*-values of all states to finite values;

  Iterate over all *s* in MDP and re-compute until convergence:

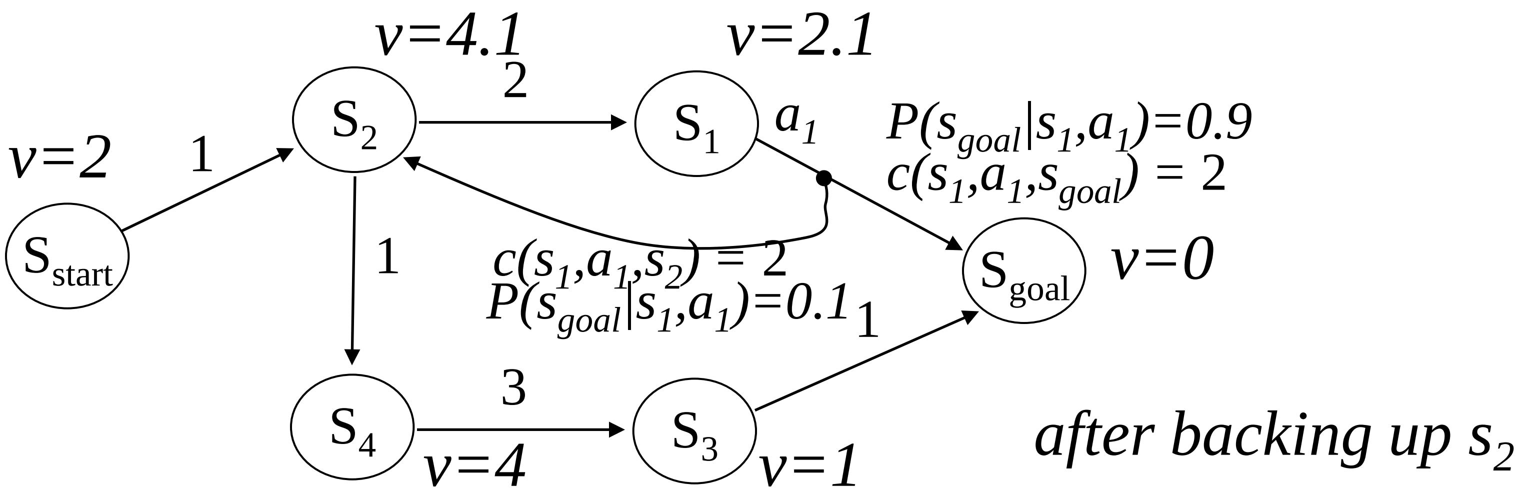
  $v(s_{goal}) = 0$

  $v(s) = \min_a E\{c(s,a,s')+v(s')\}$ *for any* $s \neq s_{goal}$

Usual convergence condition: Bellman error over all states < Δ

Bellman error: $|v(s) - \min_a E\{c(s,a,s')+v(s')\}|$ *for any* $s \neq s_{goal}$

Computing Expected Cost Minimal Plans



- Value Iteration (VI):

  Initialize *v*-values of all states to finite values;

  Iterate over all *s* in MDP and re-compute until convergence:

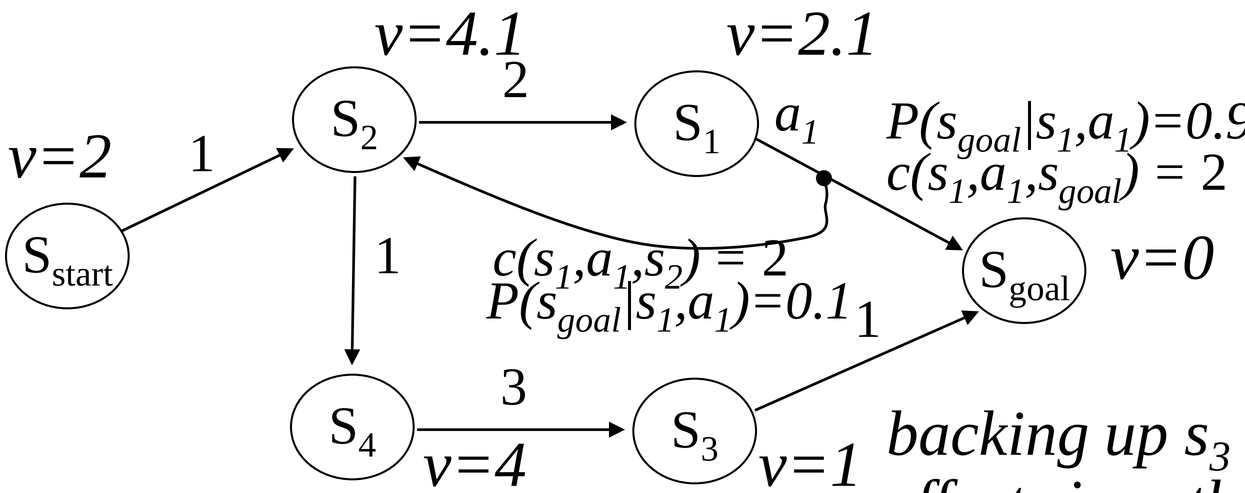
  $v(s_{goal}) = 0$

  $v(s) = \min_a E\{c(s,a,s') + v(s')\}$ *for any* $s \neq s_{goal}$

Usual convergence condition: Bellman error over all states $< \Delta$

Bellman error: $|v(s) - \min_a E\{c(s,a,s') + v(s')\}|$ *for any* $s \neq s_{goal}$

Computing Expected Cost Minimal Plans



backing up s_3 and s_4 has no effect since their Bellman errors are zero

- Value Iteration (VI):

  Initialize *v*-values of all states to finite values;

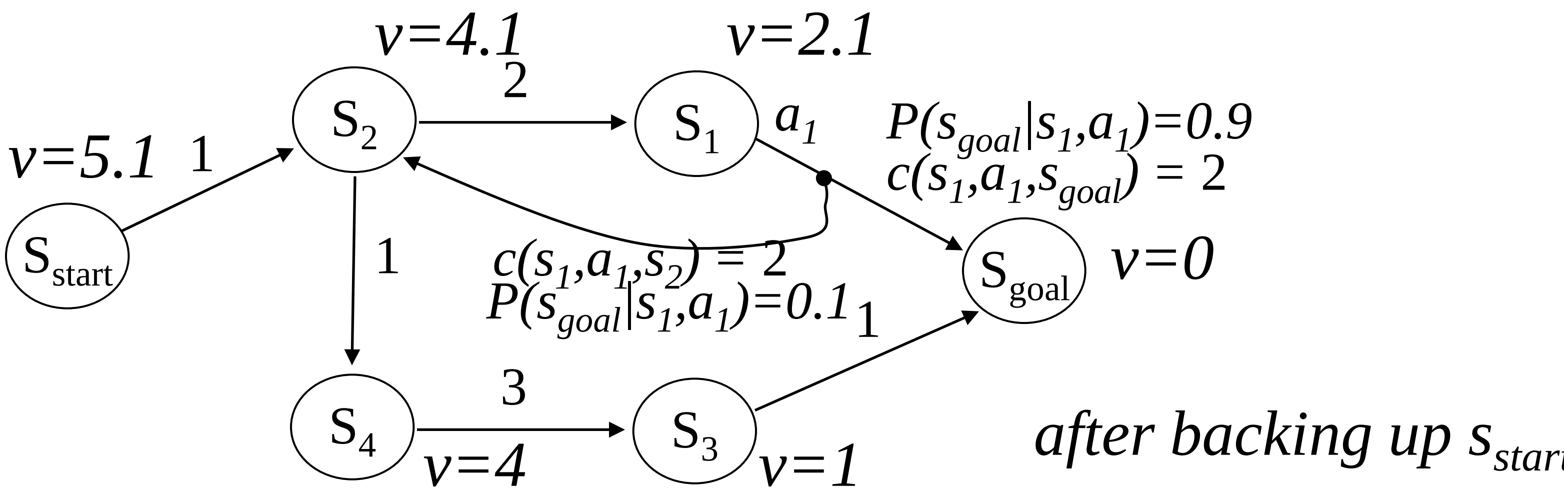
  Iterate over all *s* in MDP and re-compute until convergence:

  $$v(s_{goal}) = 0$$
  $$v(s) = \min_a E\{c(s,a,s')+v(s')\} \text{ for any } s \neq s_{goal}$$

Usual convergence condition: Bellman error over all states < Δ

Bellman error: $|v(s) - \min_a E\{c(s,a,s')+v(s')\}|$ *for any* $s \neq s_{goal}$

Computing Expected Cost Minimal Plans



- Value Iteration (VI):

  Initialize *v*-values of all states to finite values;

  Iterate over all *s* in MDP and re-compute until convergence:

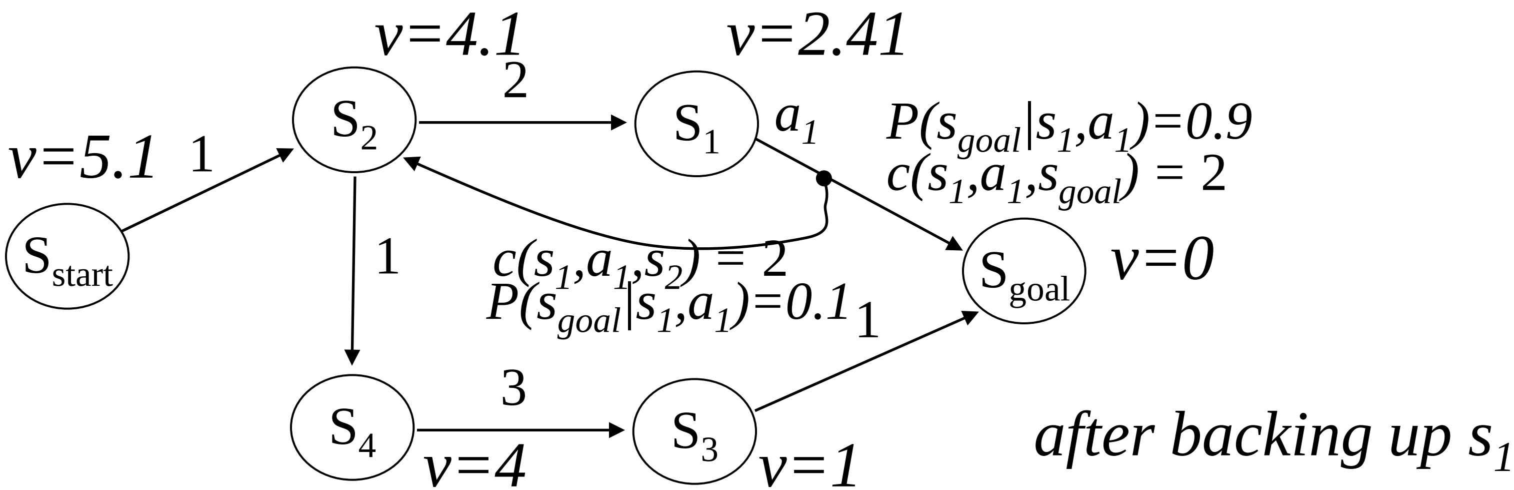
  $v(s_{goal}) = 0$

  $v(s) = \min_a E\{c(s,a,s') + v(s')\}$ *for any* $s \neq s_{goal}$

Usual convergence condition: Bellman error over all states < Δ

Bellman error: $|v(s) - \min_a E\{c(s,a,s')+v(s')\}|$ *for any* $s \neq s_{goal}$

Computing Expected Cost Minimal Plans



- Value Iteration (VI):

  Initialize *v*-values of all states to finite values;

  Iterate over all *s* in MDP and re-compute until convergence:

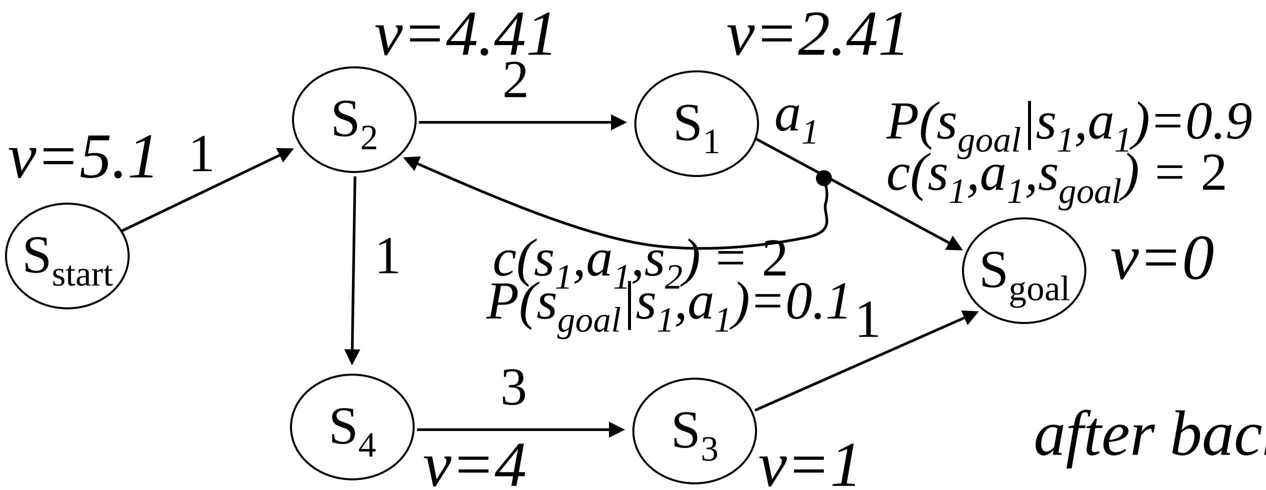
  $v(s_{goal}) = 0$

  $v(s) = \min_a E\{c(s,a,s')+v(s')\}$ *for any* $s \neq s_{goal}$

  Usual convergence condition: Bellman error over all states < Δ

  Bellman error: $|v(s) - \min_a E\{c(s,a,s')+v(s')\}|$ *for any* $s \neq s_{goal}$

Computing Expected Cost Minimal Plans



after backing up s_2

- Value Iteration (VI):

  Initialize *v*-values of all states to finite values;

  Iterate over all *s* in MDP and re-compute until convergence:

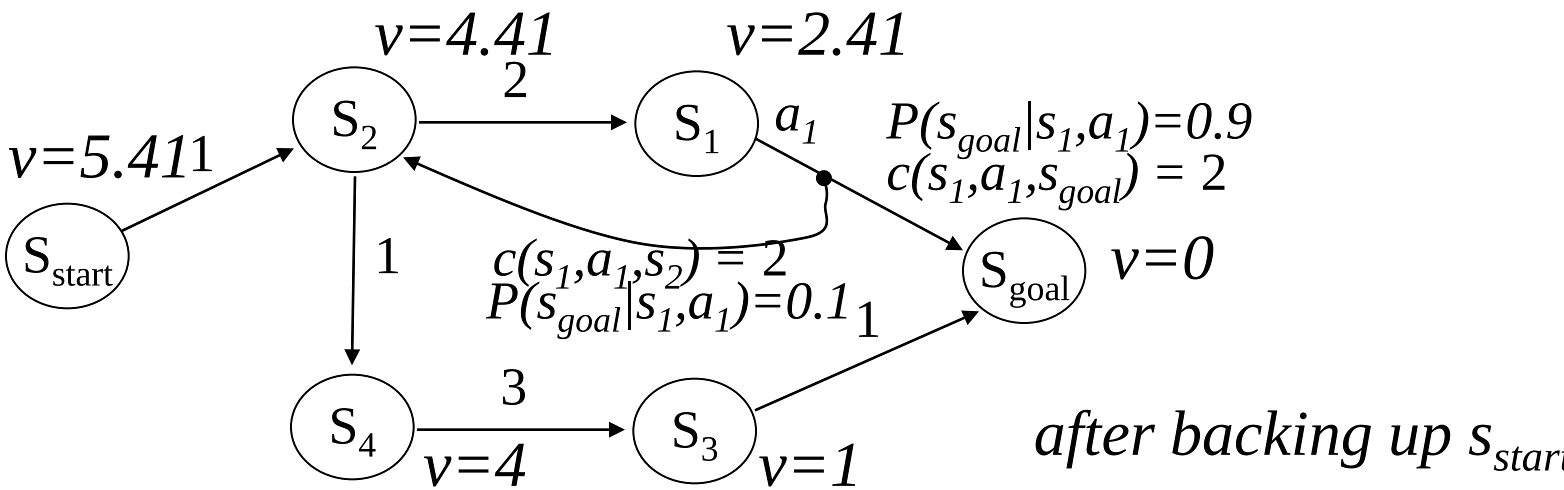
  $v(s_{goal}) = 0$

  $v(s) = \min_a E\{c(s,a,s')+v(s')\}$ *for any* $s \neq s_{goal}$

Usual convergence condition: Bellman error over all states $< \Delta$

Bellman error: $|v(s) - \min_a E\{c(s,a,s')+v(s')\}|$ *for any* $s \neq s_{goal}$

Computing Expected Cost Minimal Plans



- Value Iteration (VI):

  Initialize *v*-values of all states to finite values;

  Iterate over all *s* in MDP and re-compute until convergence:

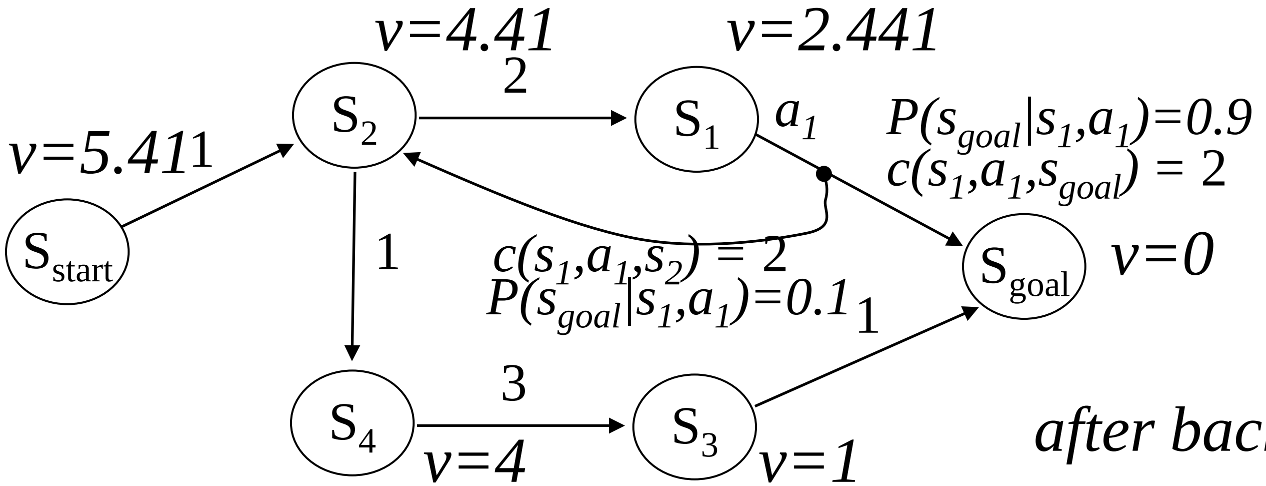
  $v(s_{goal}) = 0$

  $v(s) = \min_a E\{c(s,a,s') + v(s')\}$ *for any* $s \neq s_{goal}$

Usual convergence condition: Bellman error over all states < Δ

Bellman error: $|v(s) - \min_a E\{c(s,a,s') + v(s')\}|$ *for any* $s \neq s_{goal}$

Computing Expected Cost Minimal Plans



after backing up s_1

- Value Iteration (VI):

  Initialize *v*-values of all states to finite values;

  Iterate over all *s* in MDP and re-compute until convergence:

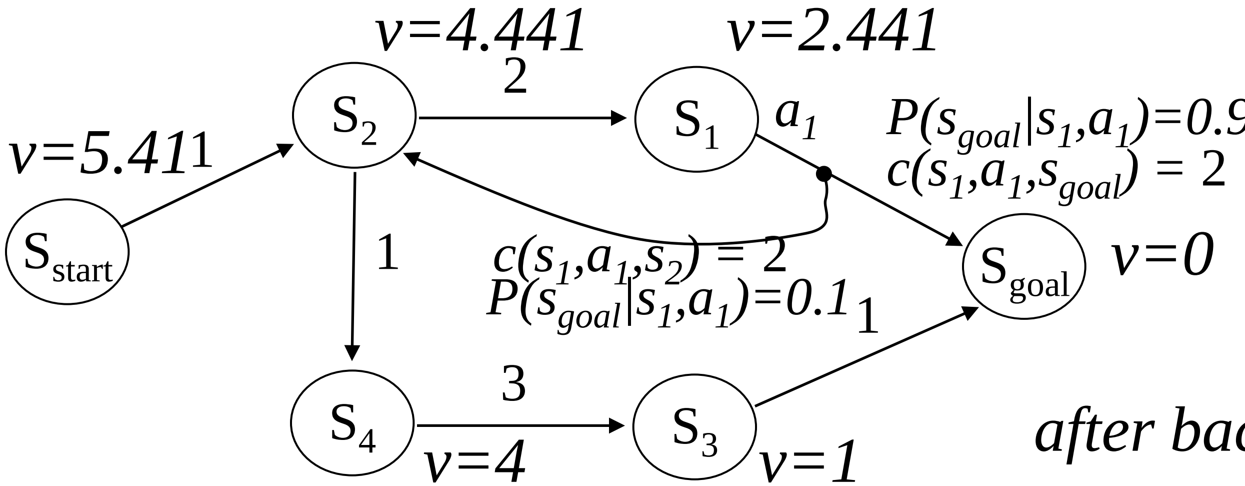
  $v(s_{goal}) = 0$

  $v(s) = \min_a E\{c(s,a,s')+v(s')\}$ *for any* $s \neq s_{goal}$

Usual convergence condition: Bellman error over all states $< \Delta$

Bellman error: $|v(s) - \min_a E\{c(s,a,s')+v(s')\}|$ *for any* $s \neq s_{goal}$

Computing Expected Cost Minimal Plans



after backing up s_2

- Value Iteration (VI):

  Initialize *v*-values of all states to finite values;

  Iterate over all *s* in MDP and re-compute until convergence:

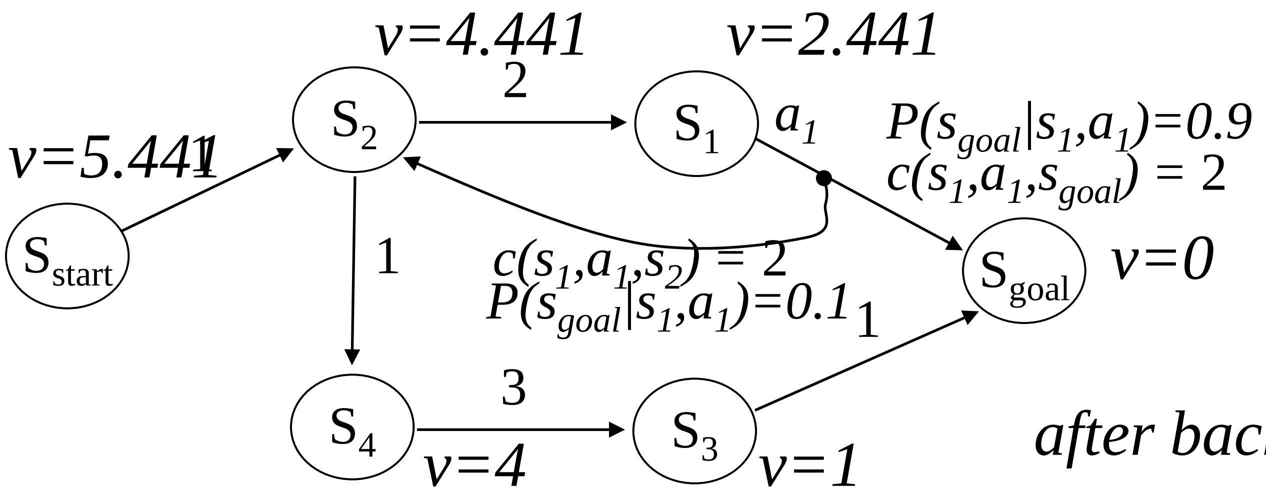
  $v(s_{goal}) = 0$

  $v(s) = \min_a E\{c(s,a,s')+v(s')\}$ *for any* $s \neq s_{goal}$

Usual convergence condition: Bellman error over all states $< \Delta$

Bellman error: $|v(s) - \min_a E\{c(s,a,s')+v(s')\}|$ *for any* $s \neq s_{goal}$

Computing Expected Cost Minimal Plans



after backing up s_{start}

- Value Iteration (VI):

Initialize *v*-values of all states to finite values;

Iterate over all *s* in MDP and re-compute until convergence:

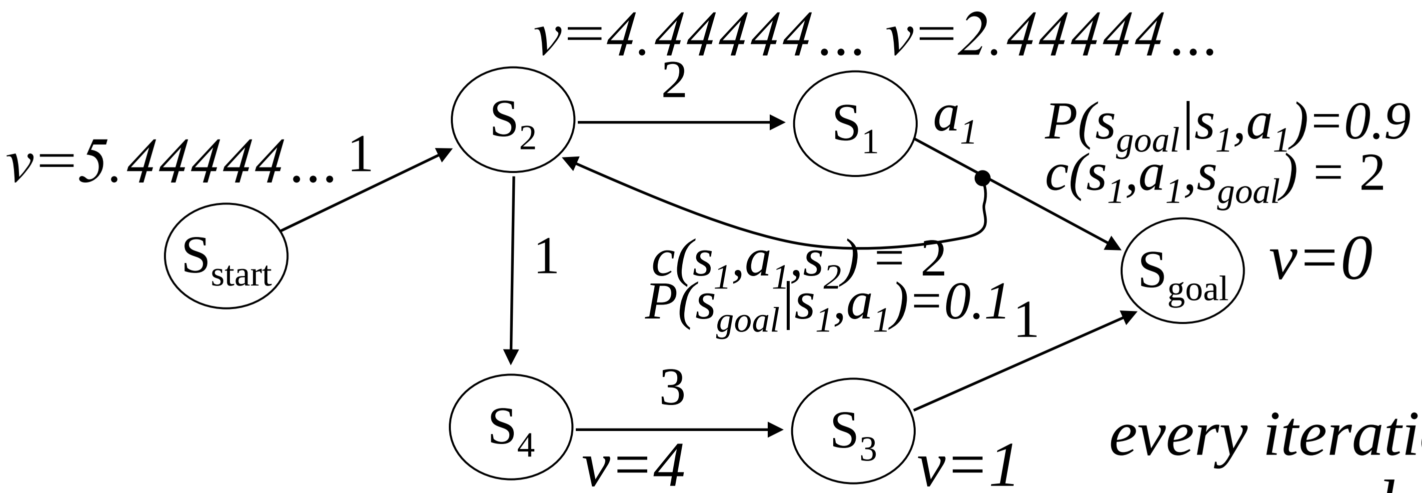
$v(s_{goal}) = 0$

$v(s) = \min_a E\{c(s,a,s')+v(s')\}$ *for any* $s \neq s_{goal}$

Usual convergence condition: Bellman error over all states $< \Delta$

Bellman error: $|v(s) - \min_a E\{c(s,a,s')+v(s')\}|$ *for any* $s \neq s_{goal}$

Computing Expected Cost Minimal Plans



every iteration computes one more decimal point

At convergence…

- Value Iteration (VI):

  Initialize *v*-values of all states to finite values;

  Iterate over all *s* in MDP and re-compute until convergence:

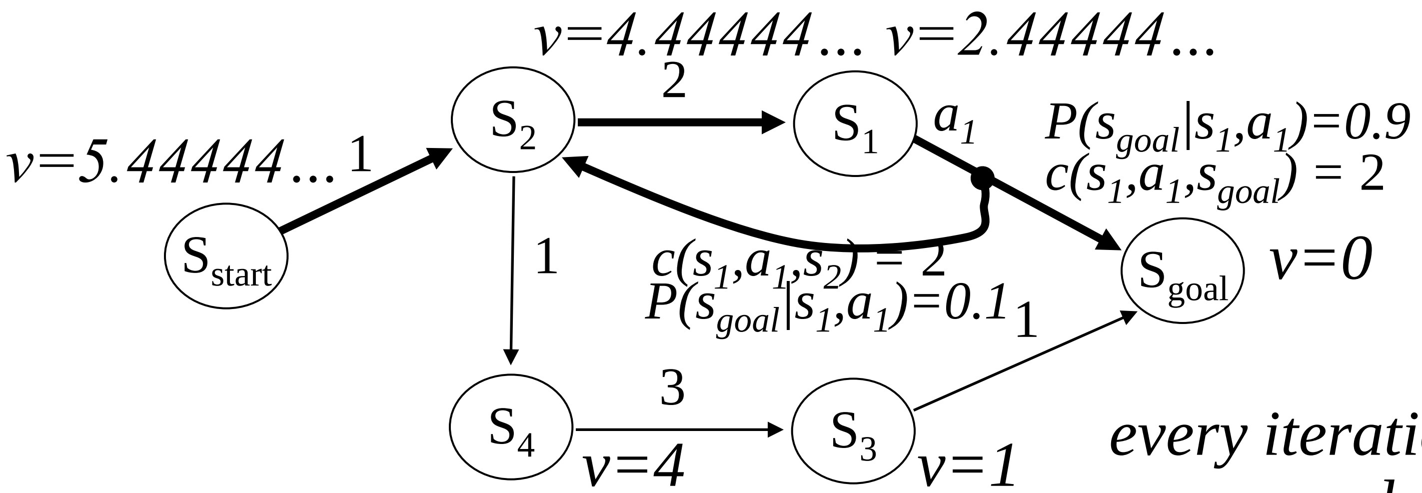
  $v(s_{goal}) = 0$

  $v(s) = \min_a E\{c(s,a,s')+v(s')\}$ *for any* $s \neq s_{goal}$

Usual convergence condition: Bellman error over all states $< \Delta$

Bellman error: $|v(s) - \min_a E\{c(s,a,s')+v(s')\}|$ *for any* $s \neq s_{goal}$

Computing Expected Cost Minimal Plans



every iteration computes one more decimal point

At convergence…

optimal policy is given by greedy policy: always select an action that minimizes $E\{c(s,a,s')+v(s')\}$

• Va…

Initialize … states to finite values;

Iterate over all *s* in MDP and re-compute until convergence:

v(…

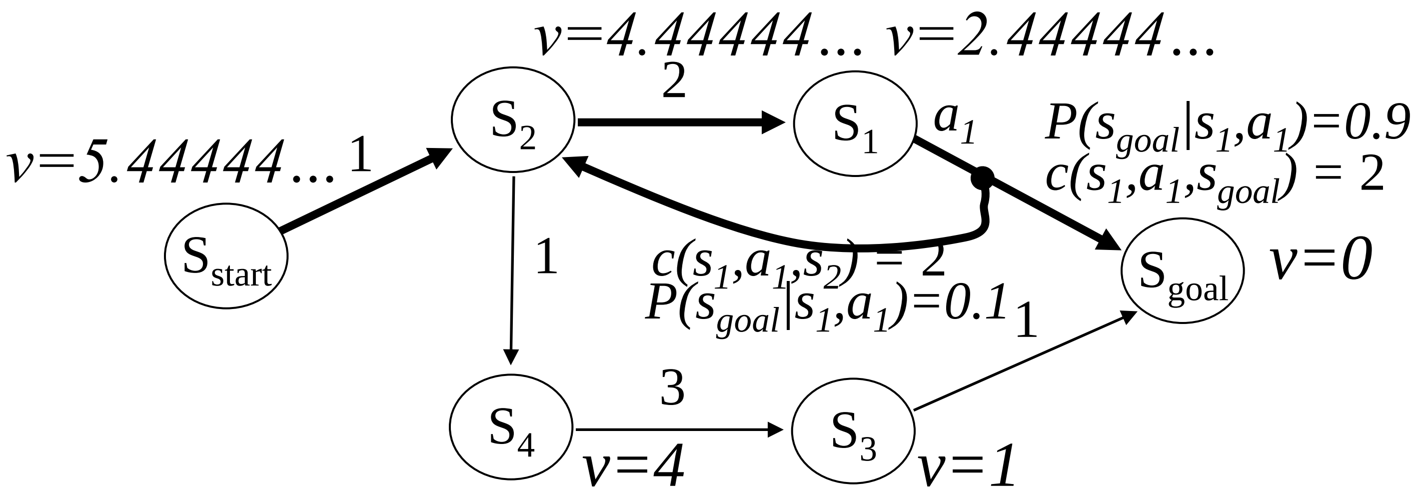
expected cost of executing greedy policy is at most: $v^*(s_{start})c_{min}/(c_{min}-\Delta)$ where c_{min} is minimum edge cost

… *for any $s \neq s_{goal}$*

Usual co… …ellman error over all states < Δ

Bellman error: $|v(s) - \min_a E\{c(s,a,s')+v(s')\}|$ for any $s \neq s_{goal}$

Computing Expected Cost Minimal Plans



- RTDP [Barto, Bradtke and Singh, 1993] (usually much much more efficient):

Initialize v-values of all states to admissible values;

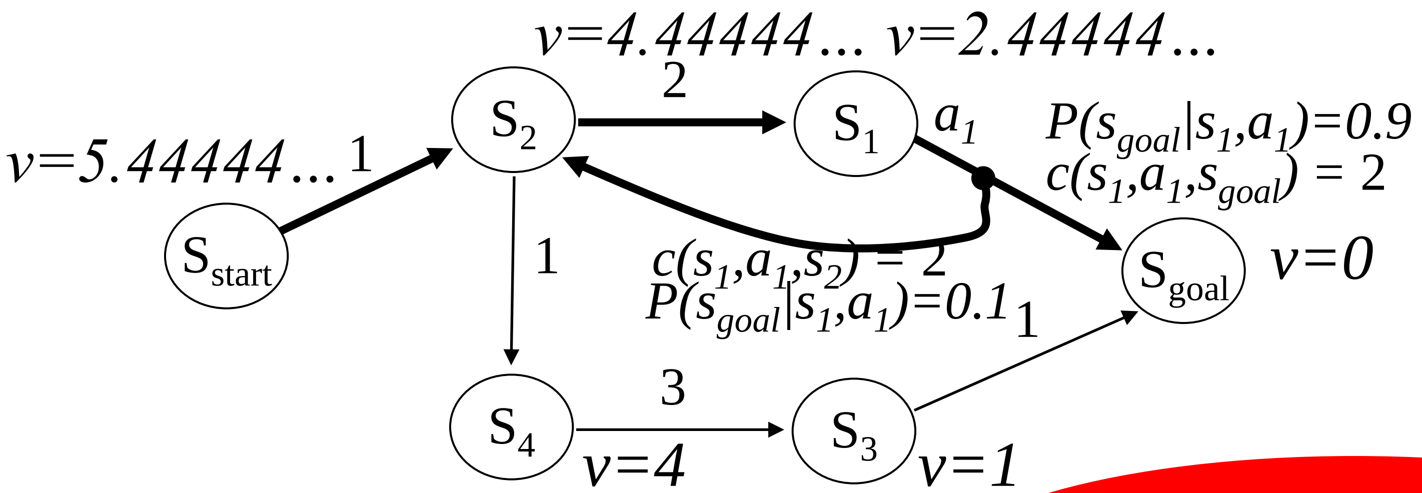
1. Follow greedy policy picking outcomes at random until goal is reached;

2. Backup all states visited on the way;

3. Reset to s_{start} and repeat 1-3 until all states on the current greedy policy have Bellman errors < Δ;

Computing Expected Cost Minimal Plans

Maxim



RTDP focusses its backups on what is relevant to the optimal plan rather than computing ALL state values (like VI)

- RTDP [Barto, Bradtke and Singh, 1993] (usually much more efficient):

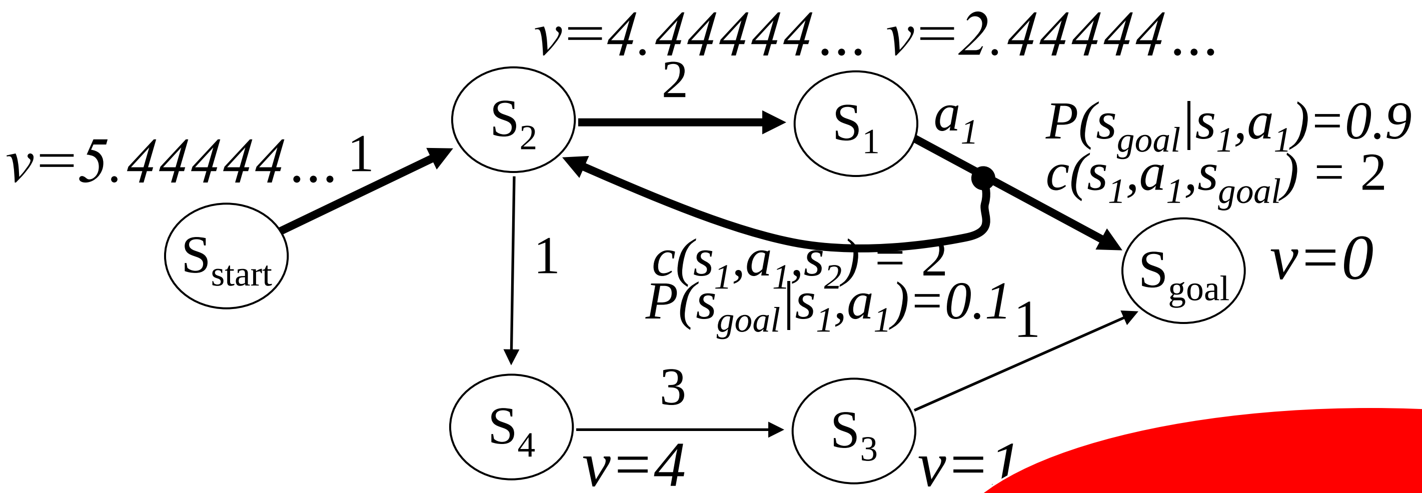
  Initialize v-values of all states to admissible values;

  1. Follow greedy policy picking outcomes at random until goal is reached;

  2. Backup all states visited on the way;

  3. Reset to s_{start} and repeat 1-3 until all states on the current greedy policy have Bellman errors < Δ;

Computing Expected Cost Minimal Plans



expected cost of executing greedy policy is at most:
$v^*(s_{start})c_{min}/(c_{min}-\Delta)$
where c_{min} is minimum edge cost

- RTDP [Barto, Bradtke and Singh, 1993] (usually much much more efficient):

  Initialize v-values of all states to admissible values;

  1. Follow greedy policy picking outcomes at random until goal is reached;

  2. Backup all states visited on the way;

  3. Reset to s_{start} and repeat 1-3 until all states on the current greedy policy have Bellman errors < Δ;

Table of Contents

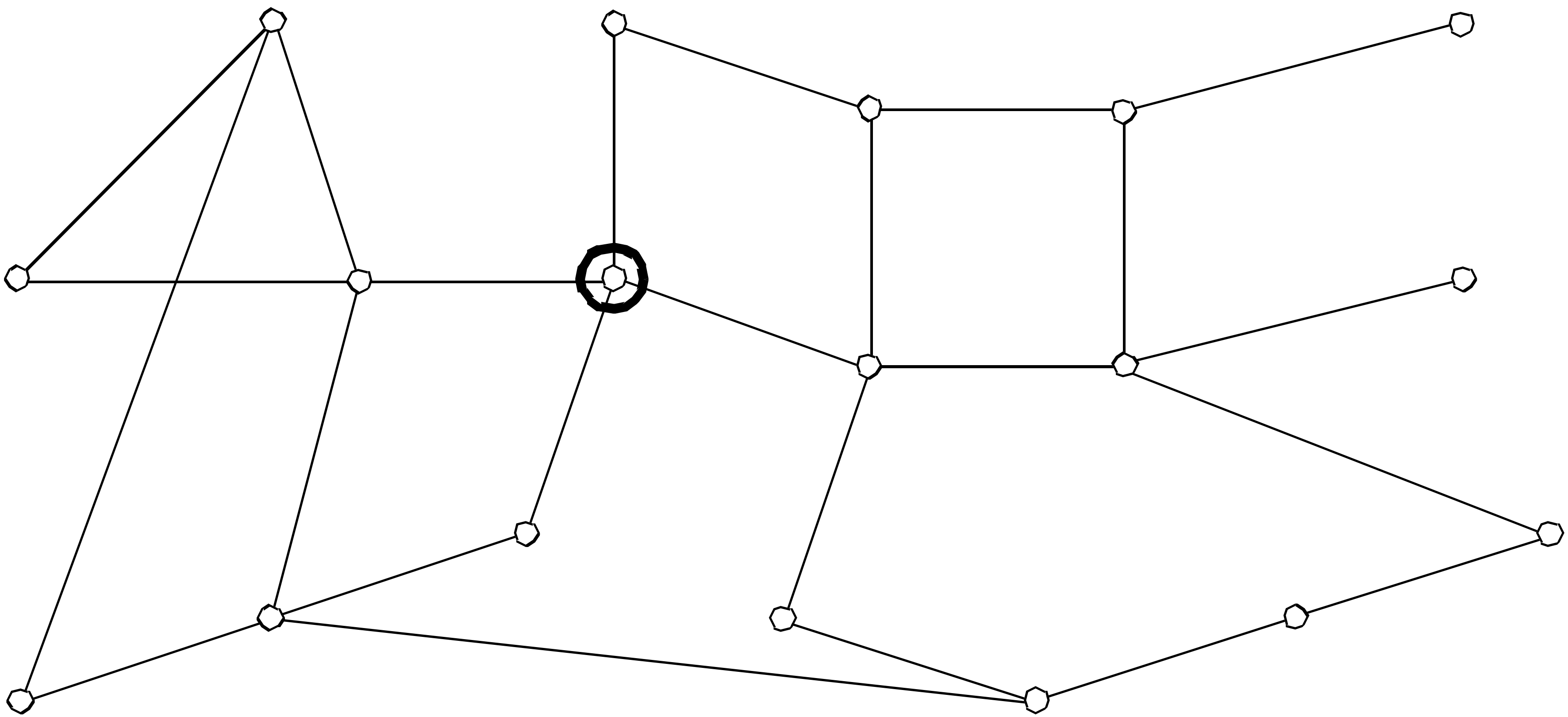
Planning Problems and Strategies

- Greedy Agent-Centered Search
- Three Robot-Navigation Problems and Approaches
  - Localization using Agent-Centered Search: Greedy Localization
  - Mapping using Agent-Centered Search: Greedy Mapping
  - Stationary Target Search in Unknown Terrain using Assumption-Based Planning: Planning with the Freespace Assumption
- Summary
  - Agent-Centered Search
  - Planning with the Freespace Assumption
  - Real-Time Search

Greedy Agent-Centered Search

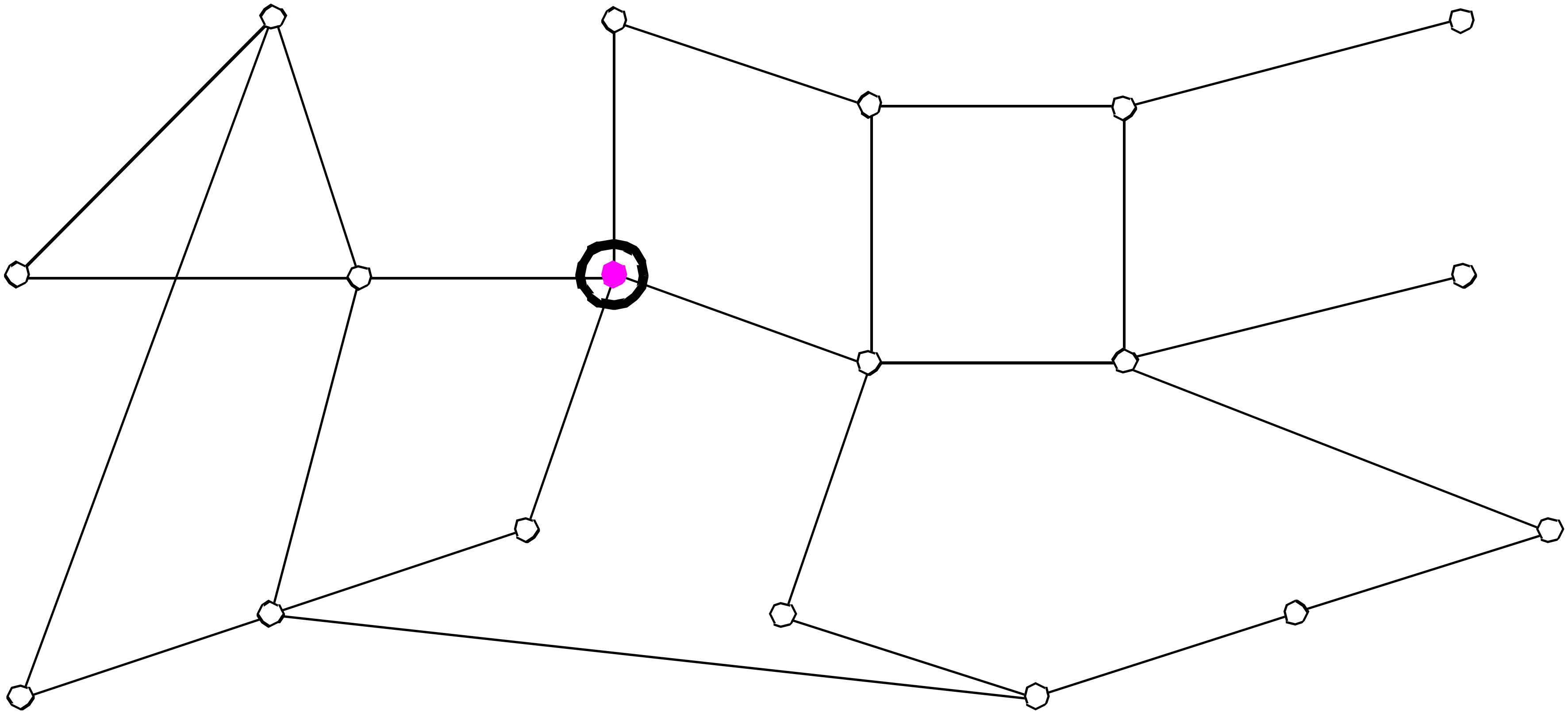
- Greedy agent-centered search starts at some state. It marks the robot state (and perhaps other states as well) as uninteresting and then moves to the closest interesting state. It repeats the process until all states are marked uninteresting.

Greedy Agent-Centered Search



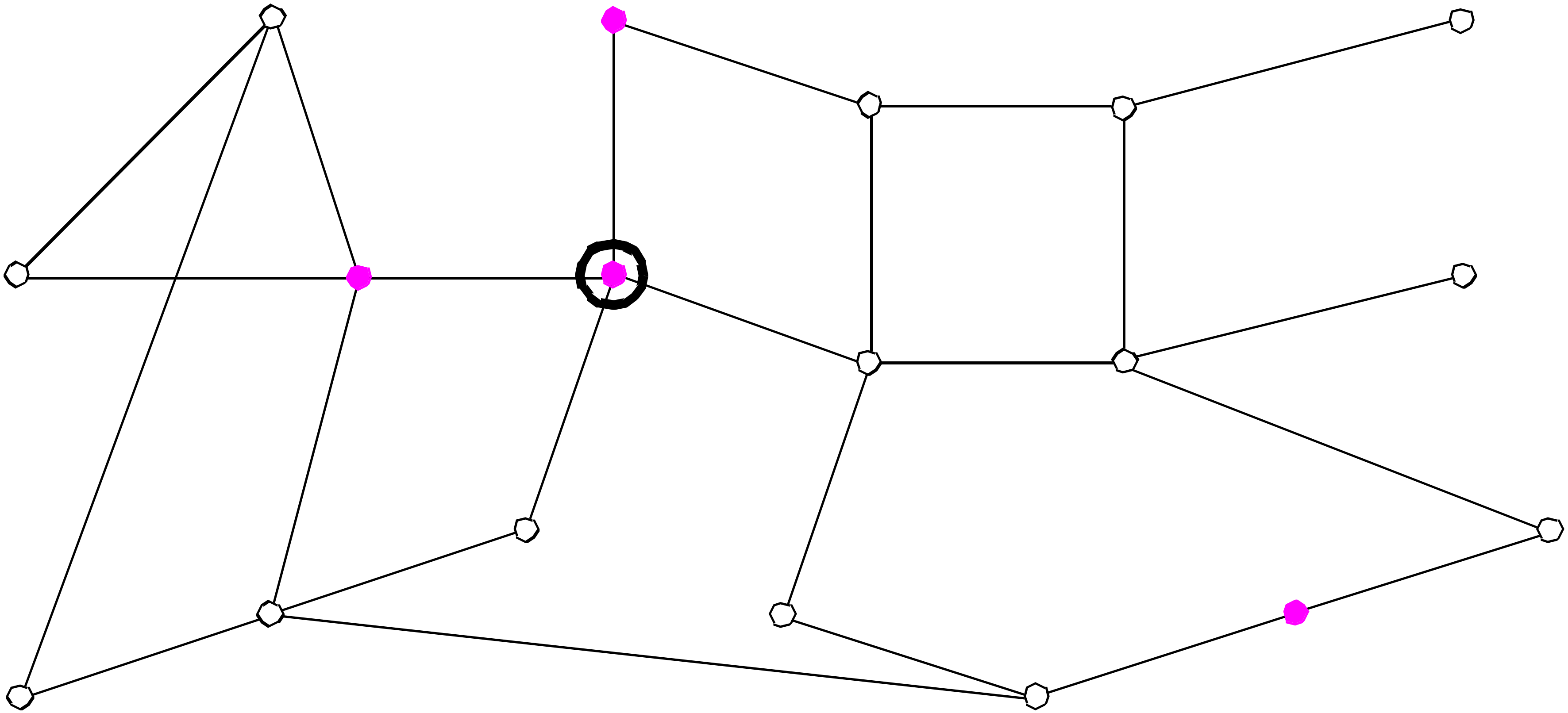
number of movements = 0

Greedy Agent-Centered Search



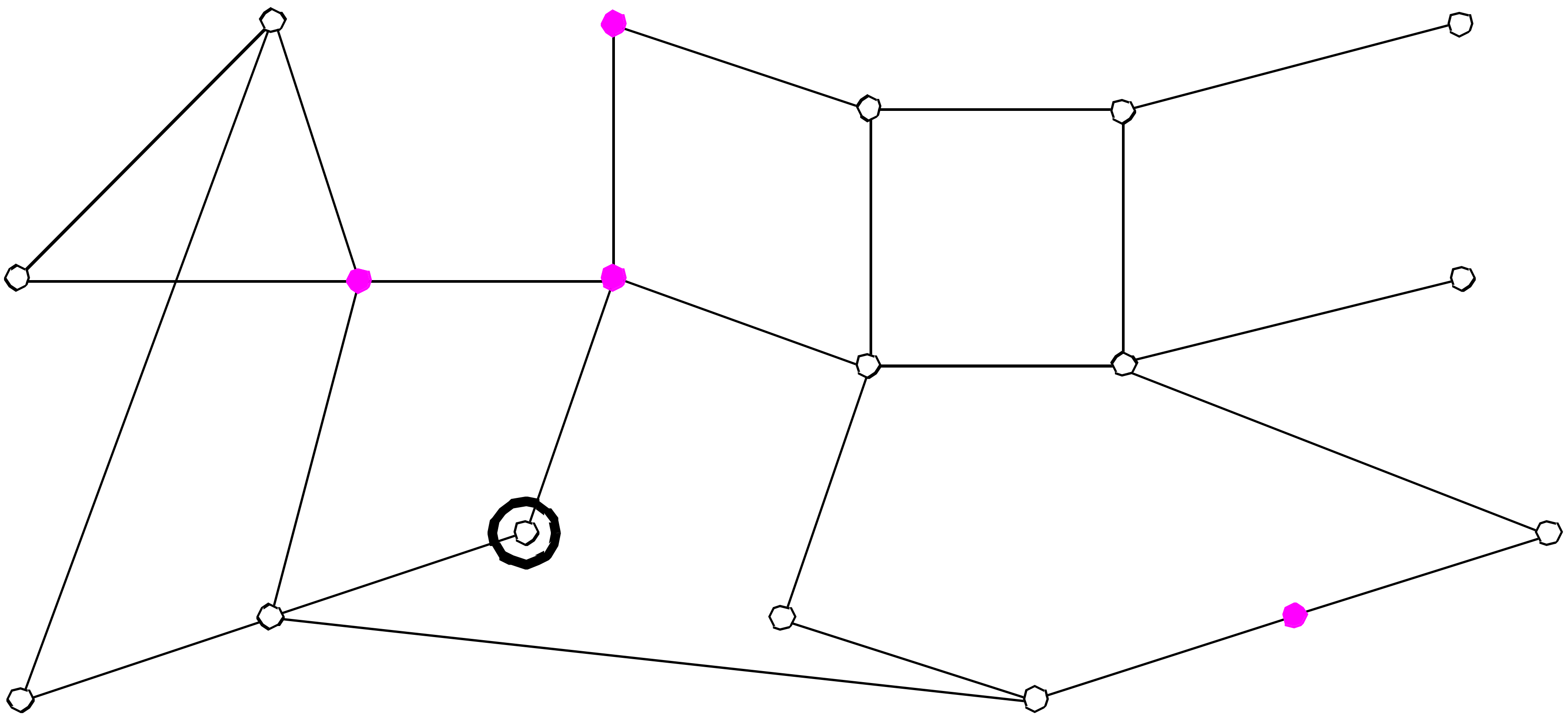
number of movements = 0

Greedy Agent-Centered Search



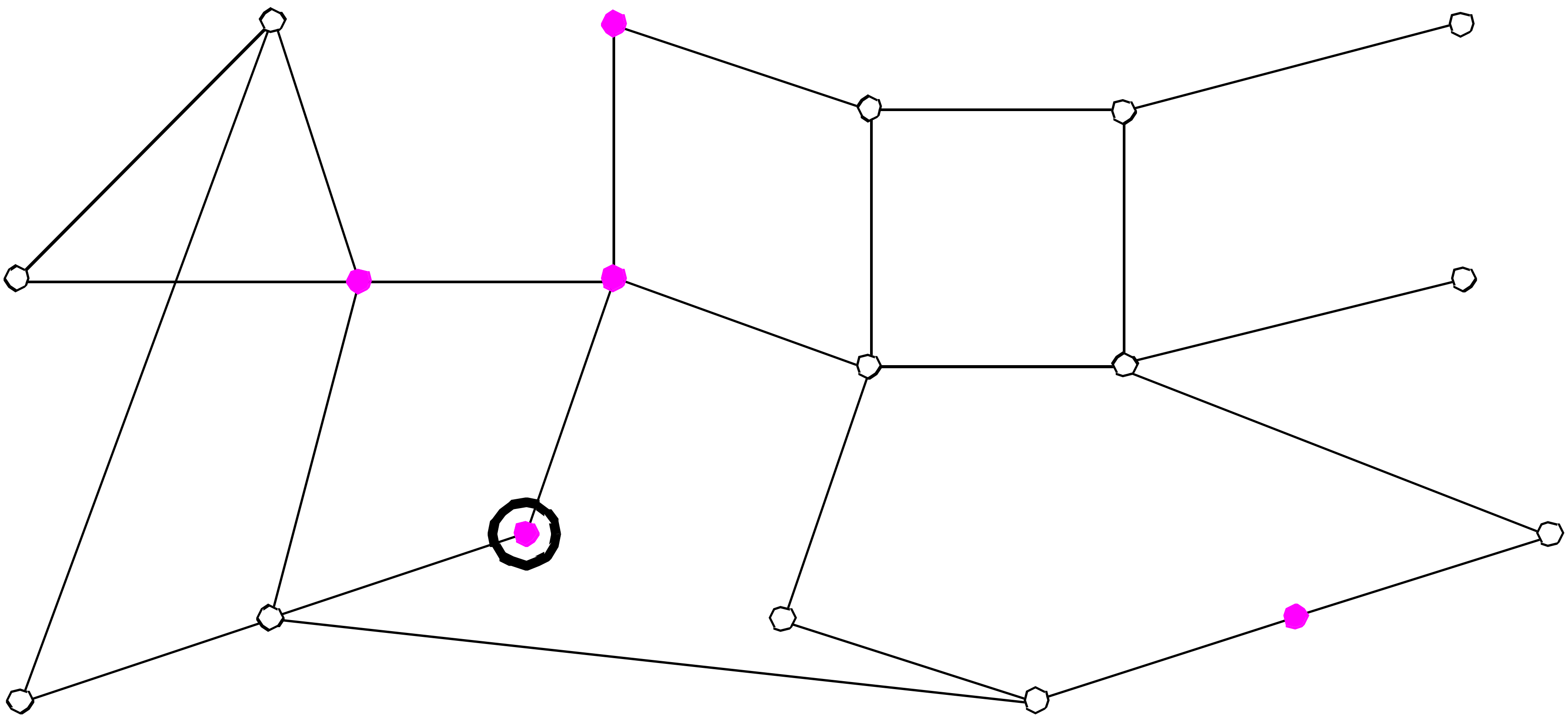
number of movements = 0

Greedy Agent-Centered Search



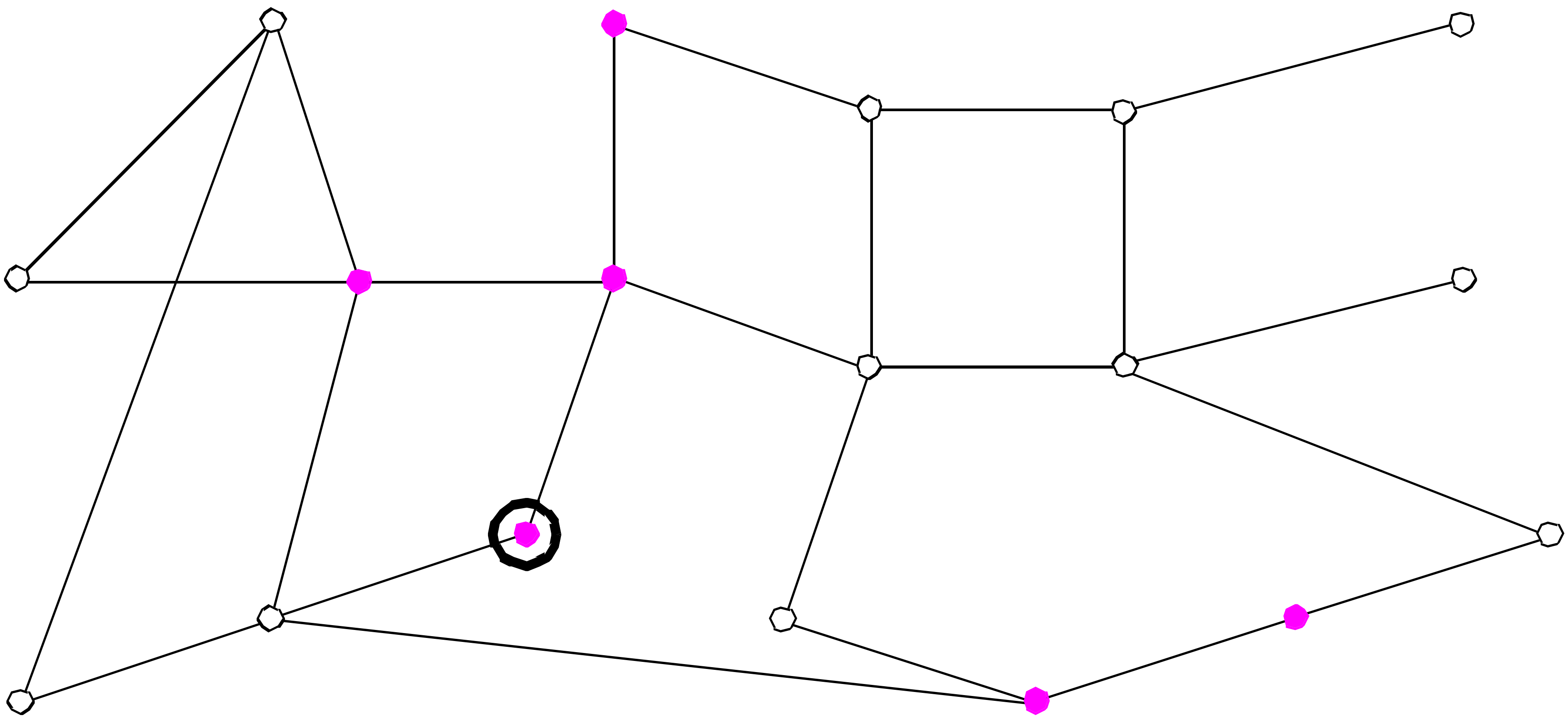
number of movements = 1

Greedy Agent-Centered Search



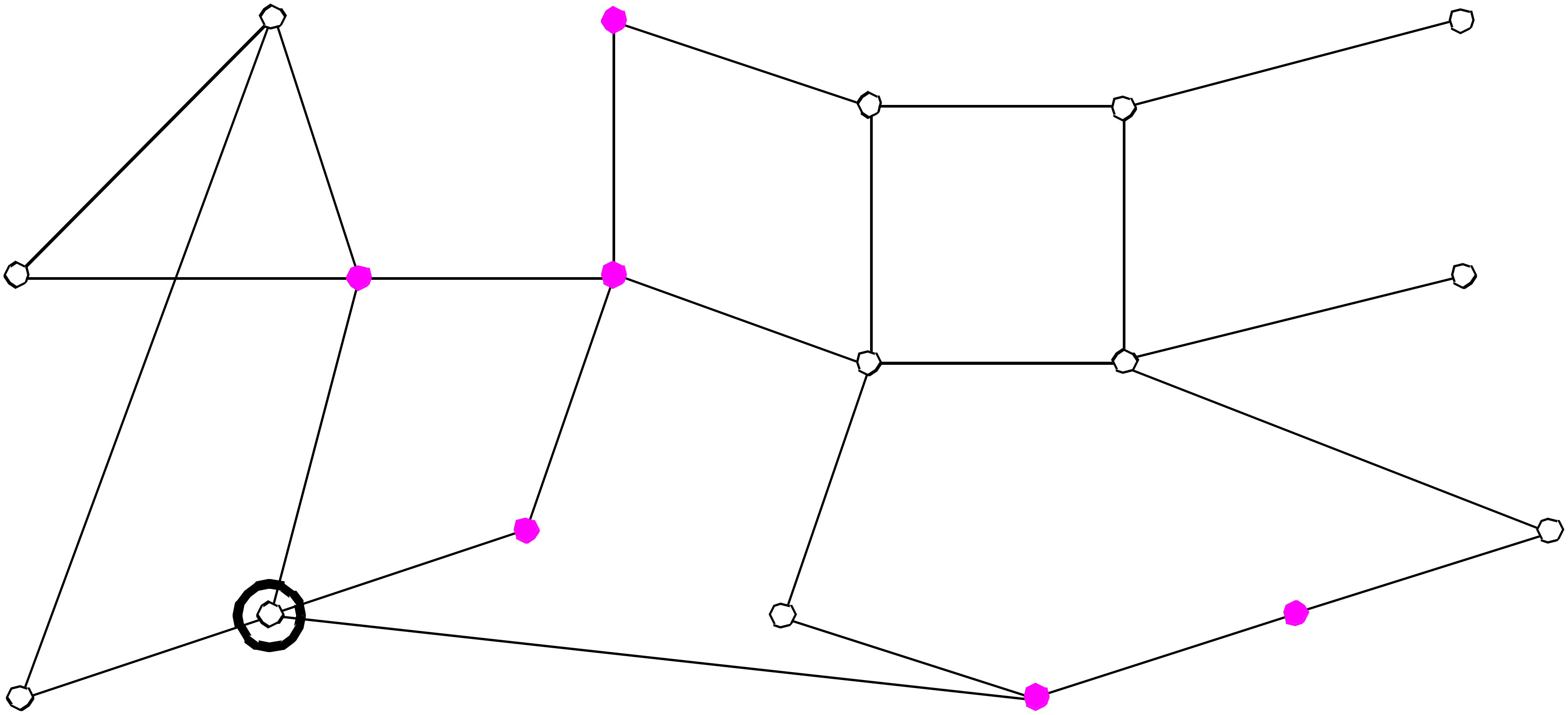
number of movements = 1

Greedy Agent-Centered Search



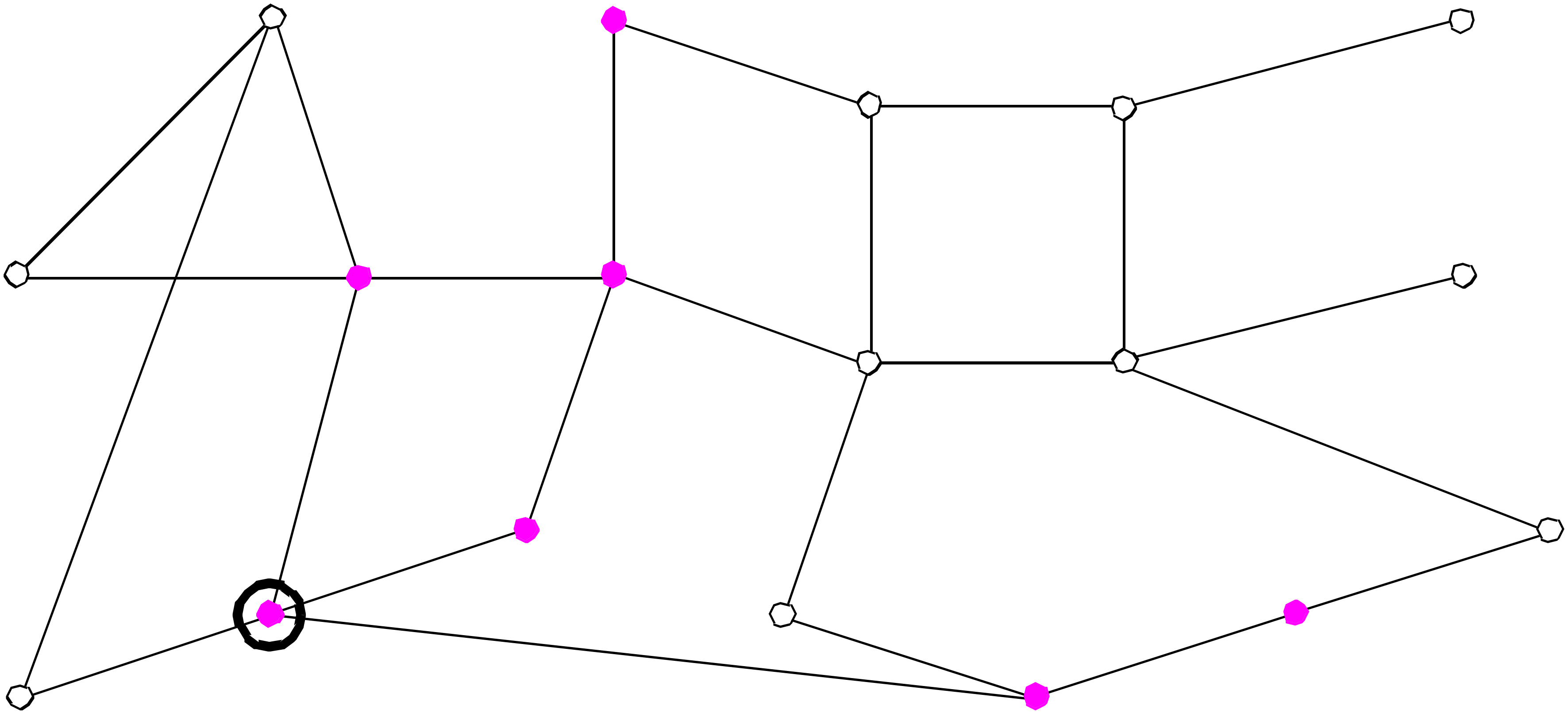
number of movements = 1

Greedy Agent-Centered Search



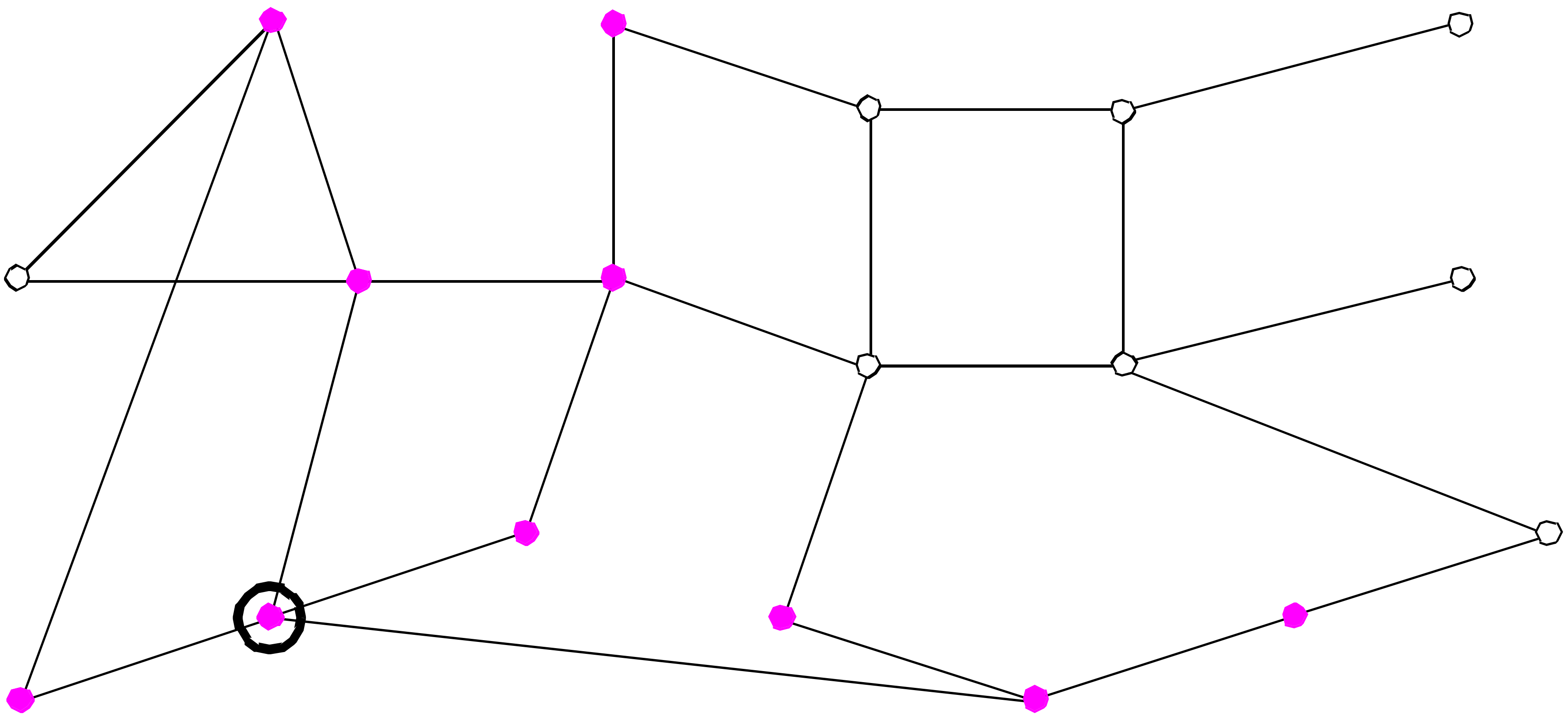
number of movements = 2

Greedy Agent-Centered Search



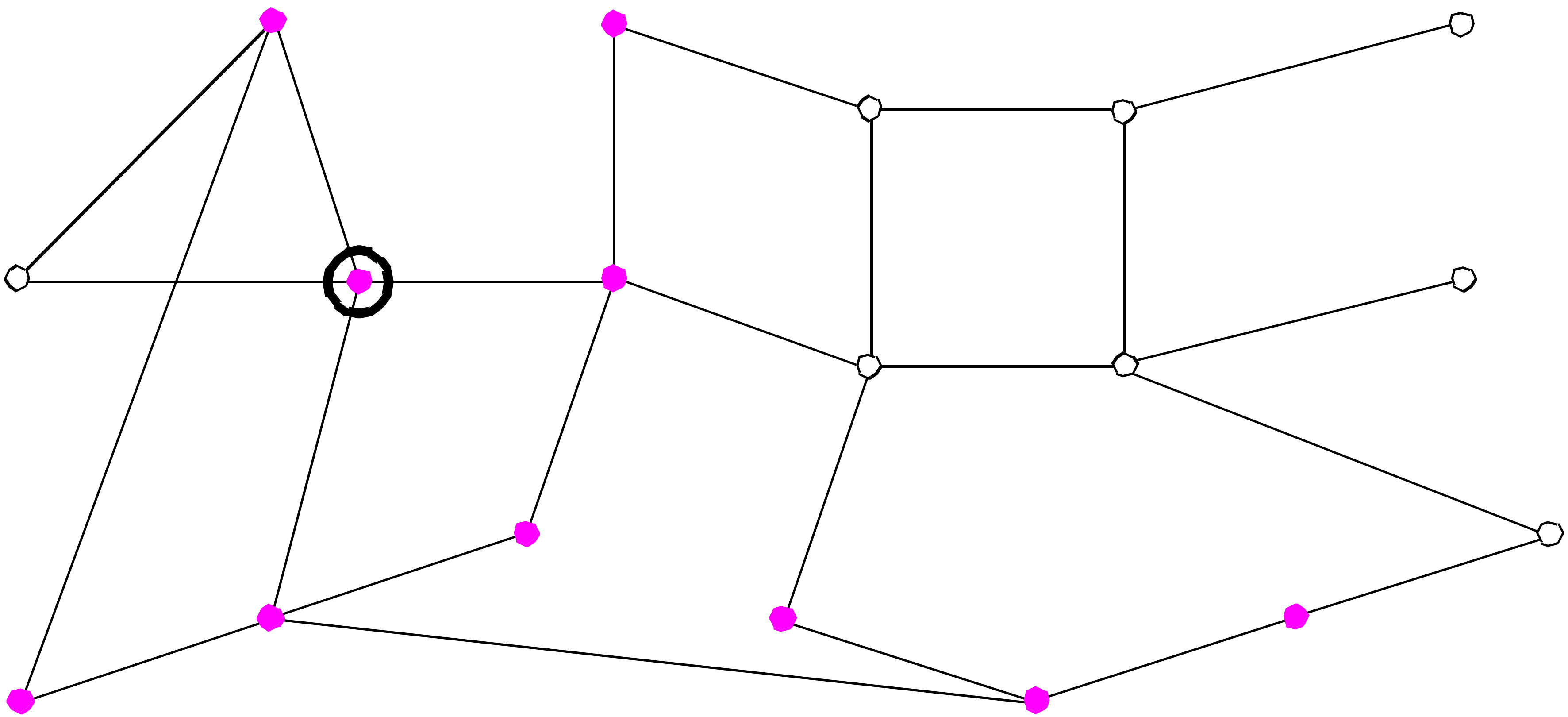
number of movements = 2

Greedy Agent-Centered Search



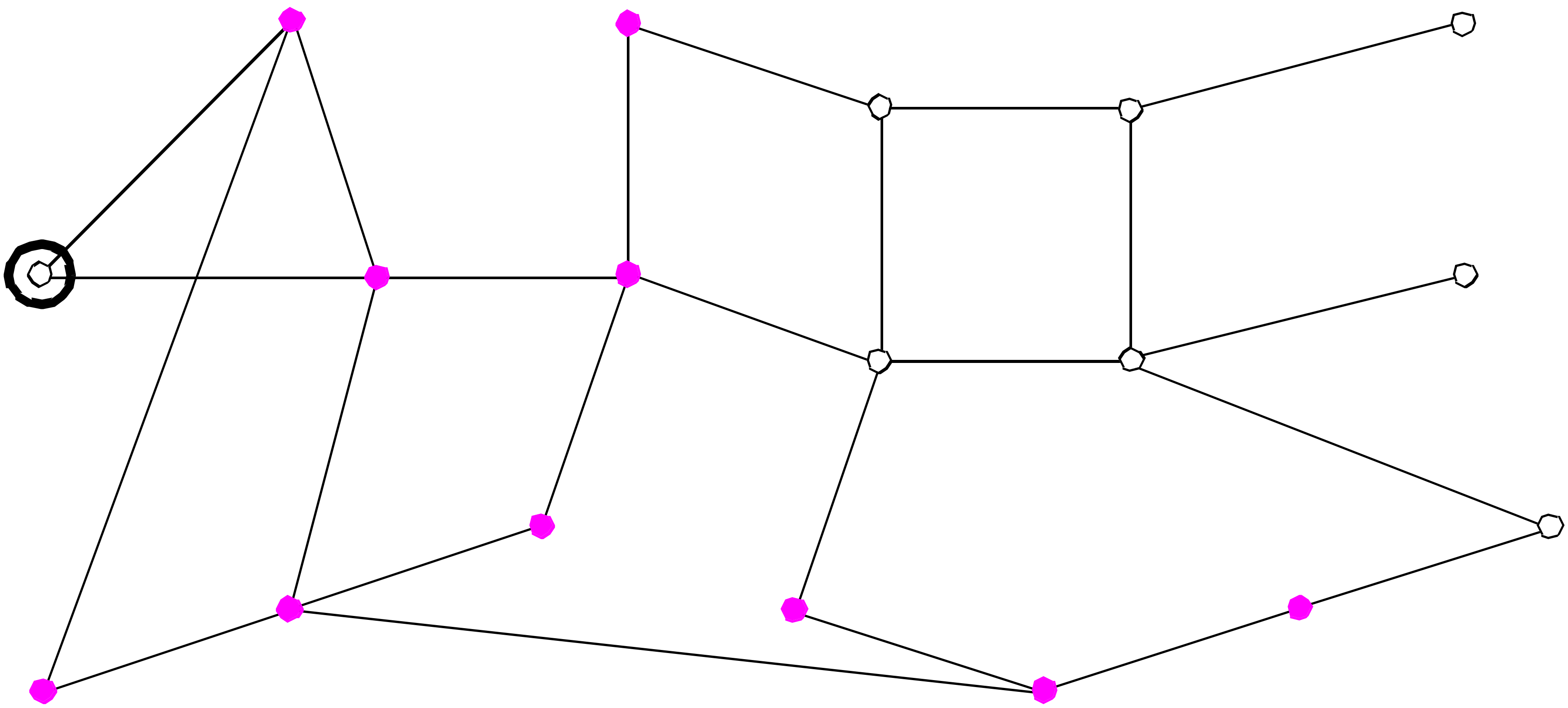
number of movements = 2

Greedy Agent-Centered Search



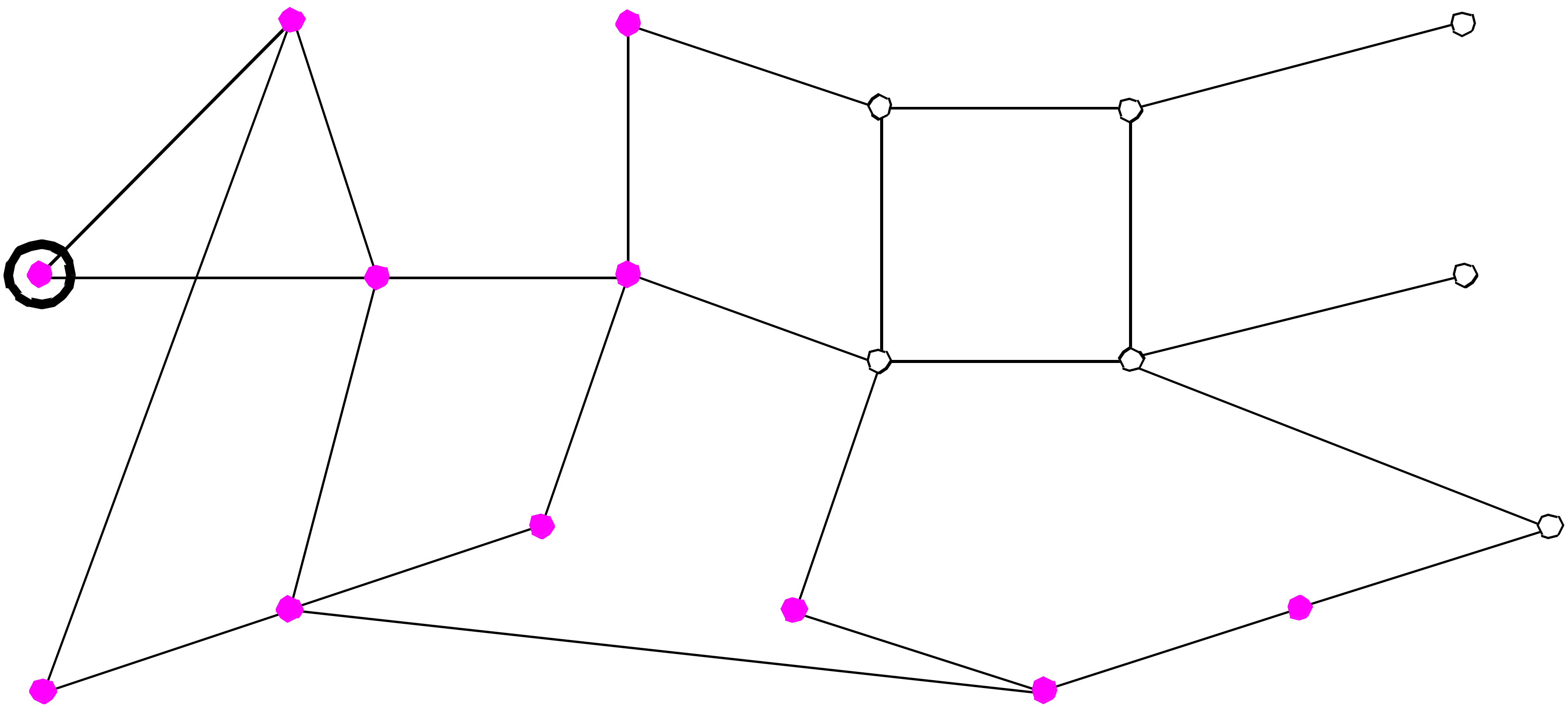
number of movements = 3

Greedy Agent-Centered Search



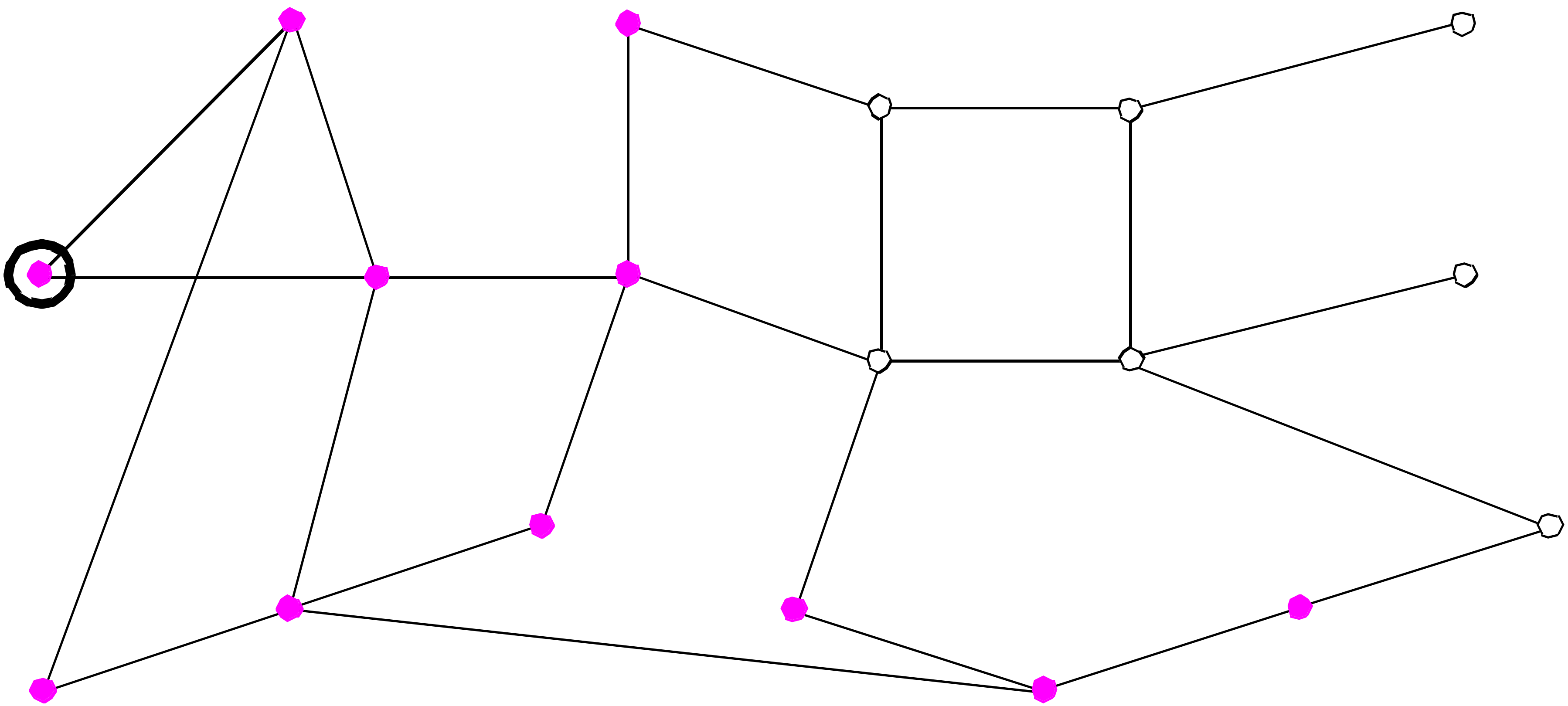
number of movements = 4

Greedy Agent-Centered Search



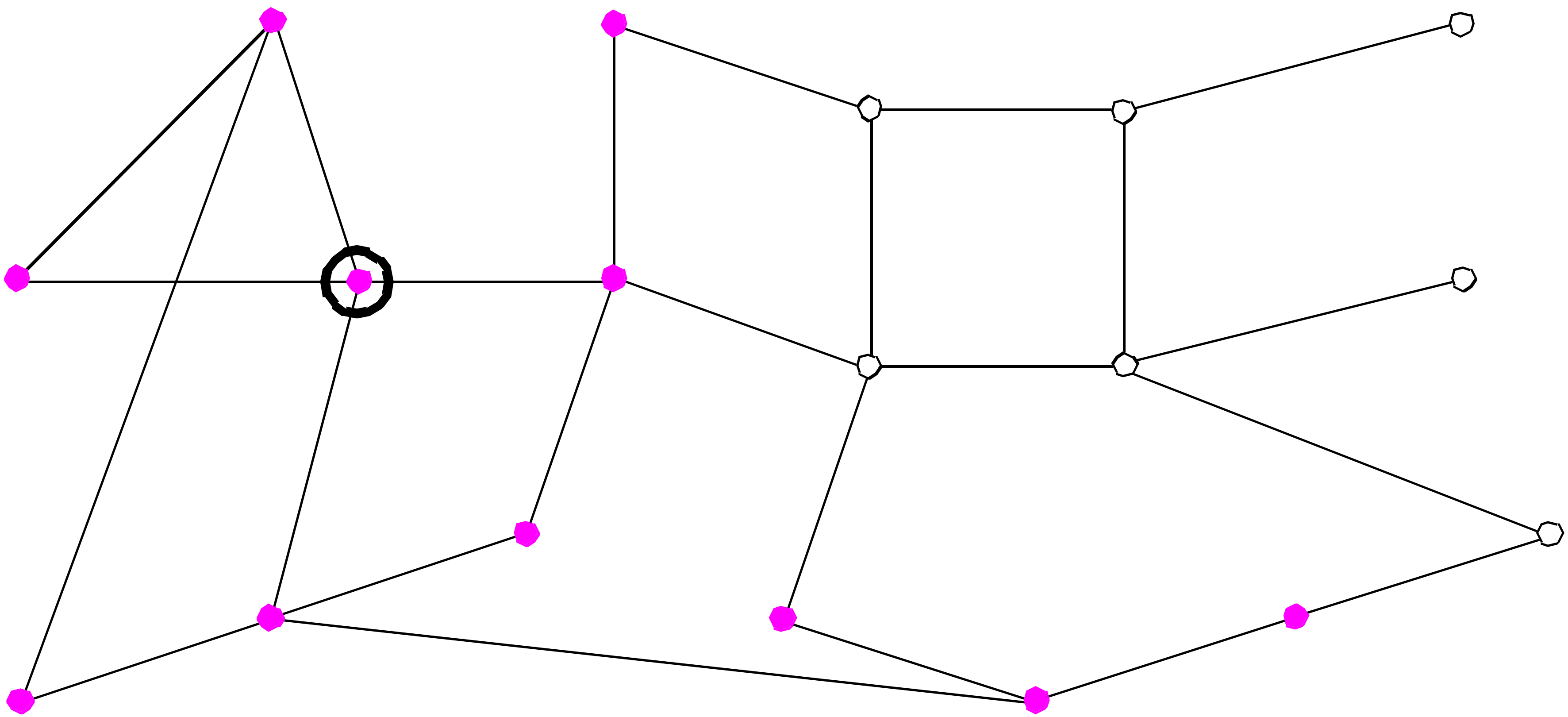
number of movements = 4

Greedy Agent-Centered Search



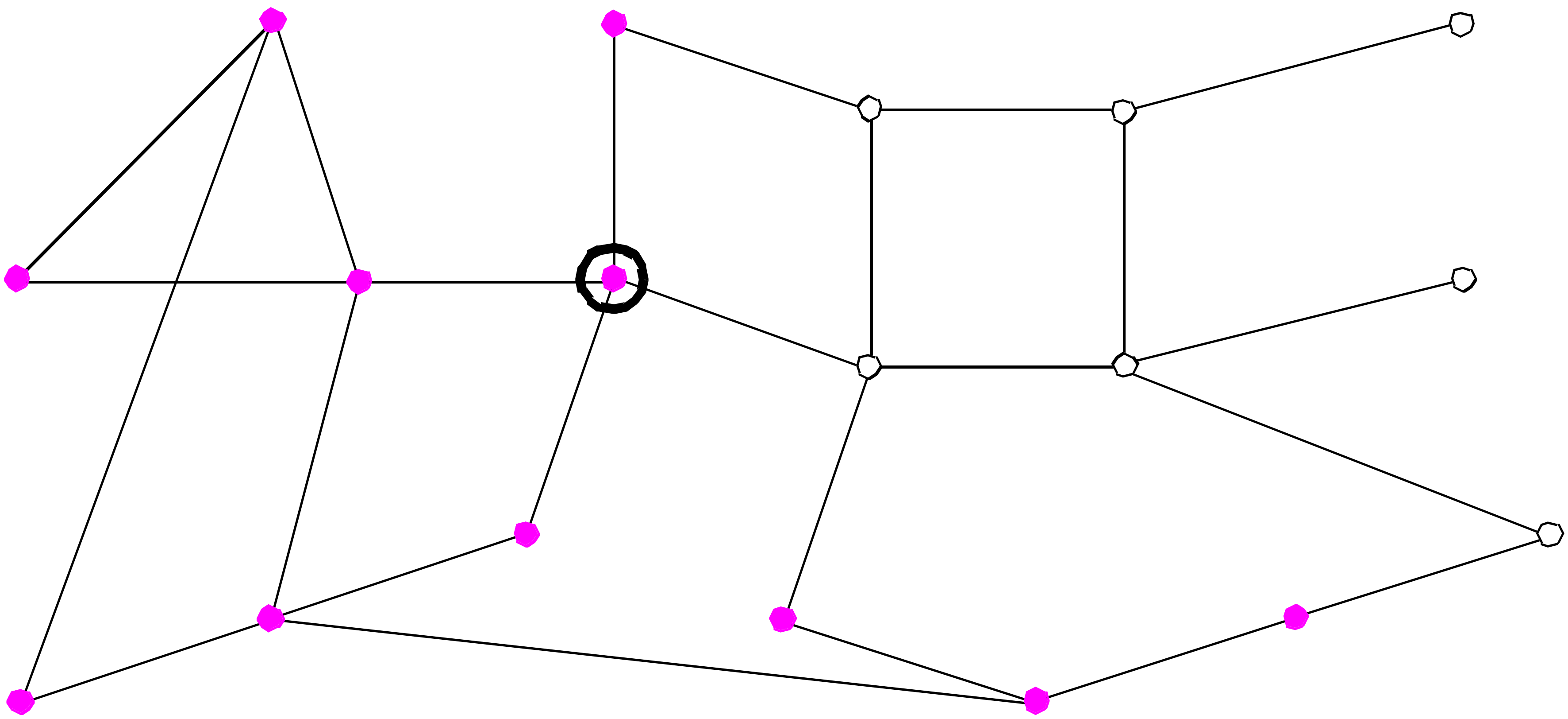
number of movements = 4

Greedy Agent-Centered Search



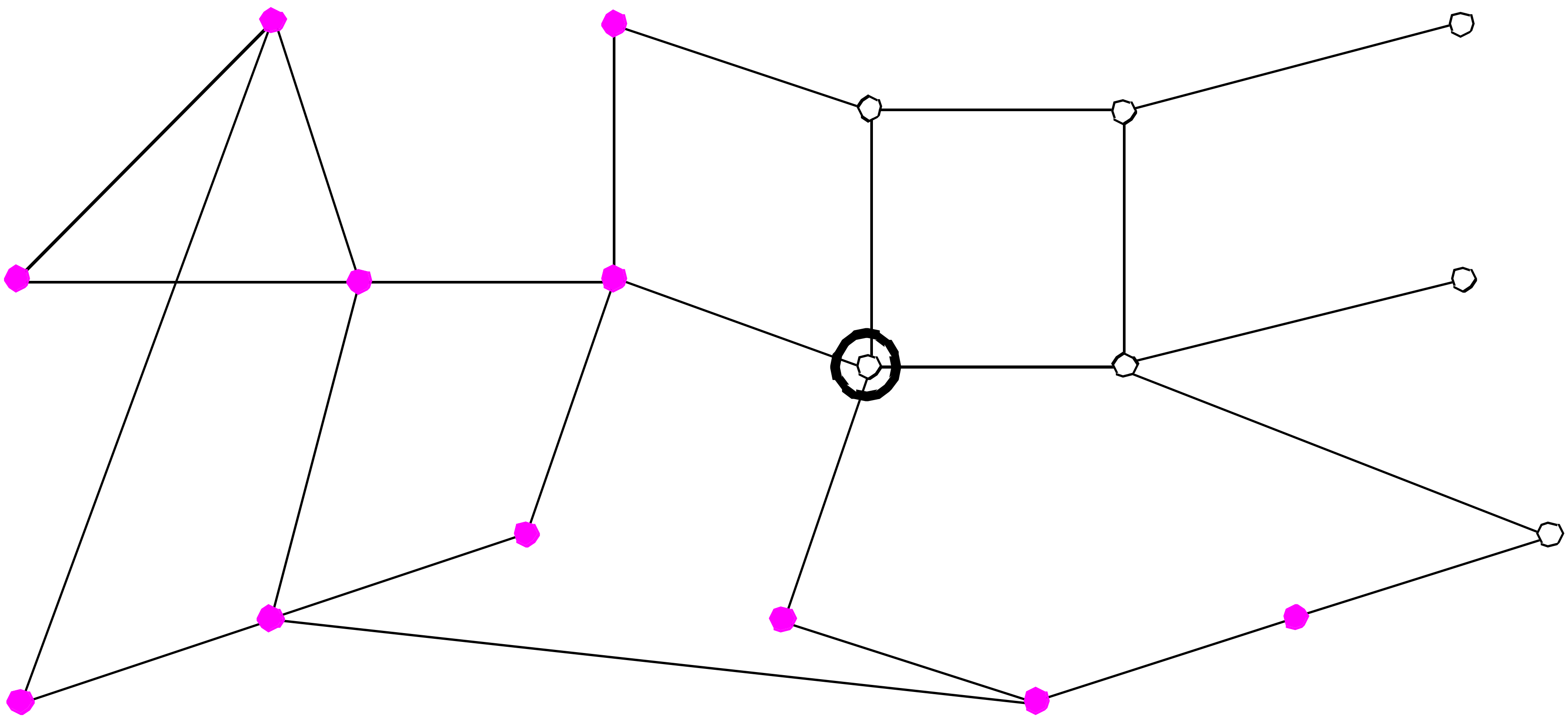
number of movements = 5

Greedy Agent-Centered Search



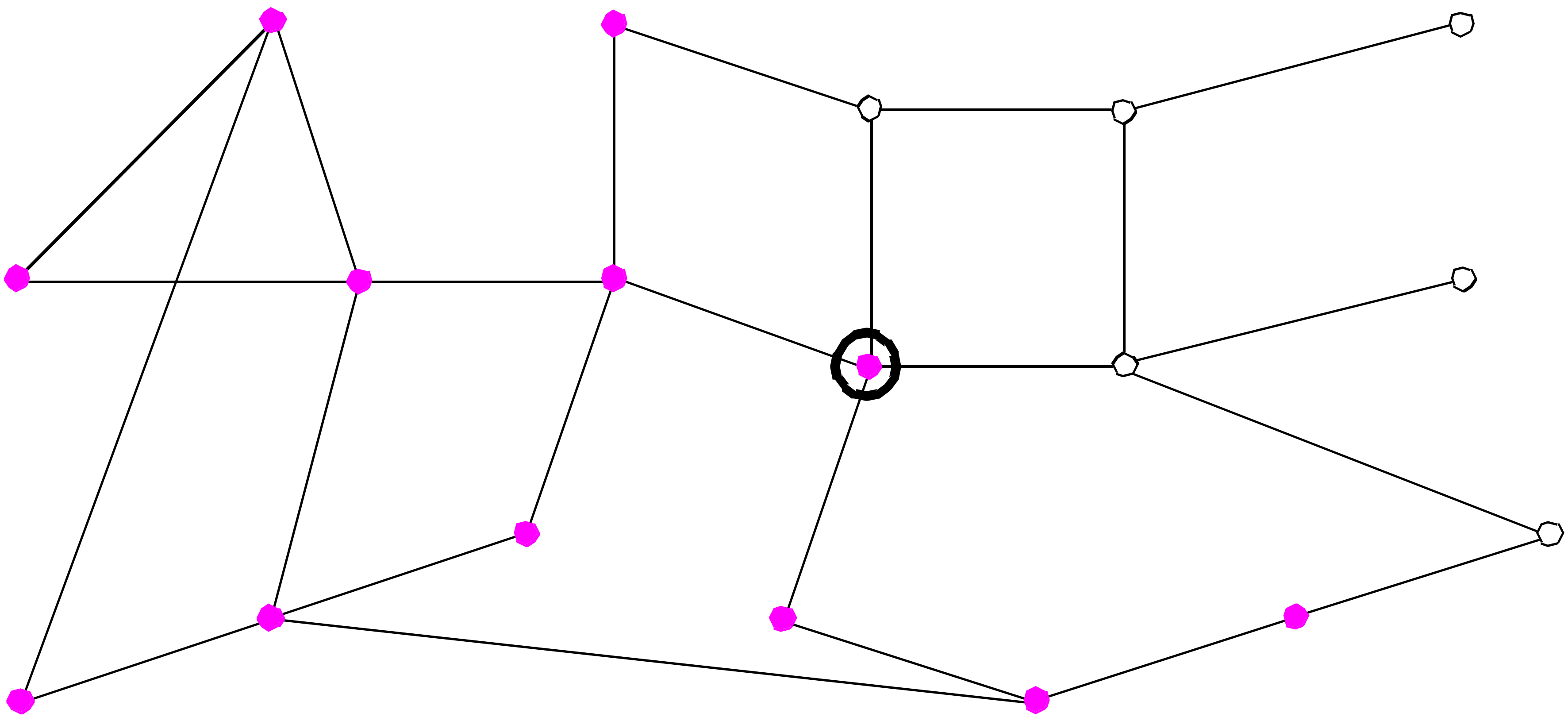
number of movements = 6

Greedy Agent-Centered Search



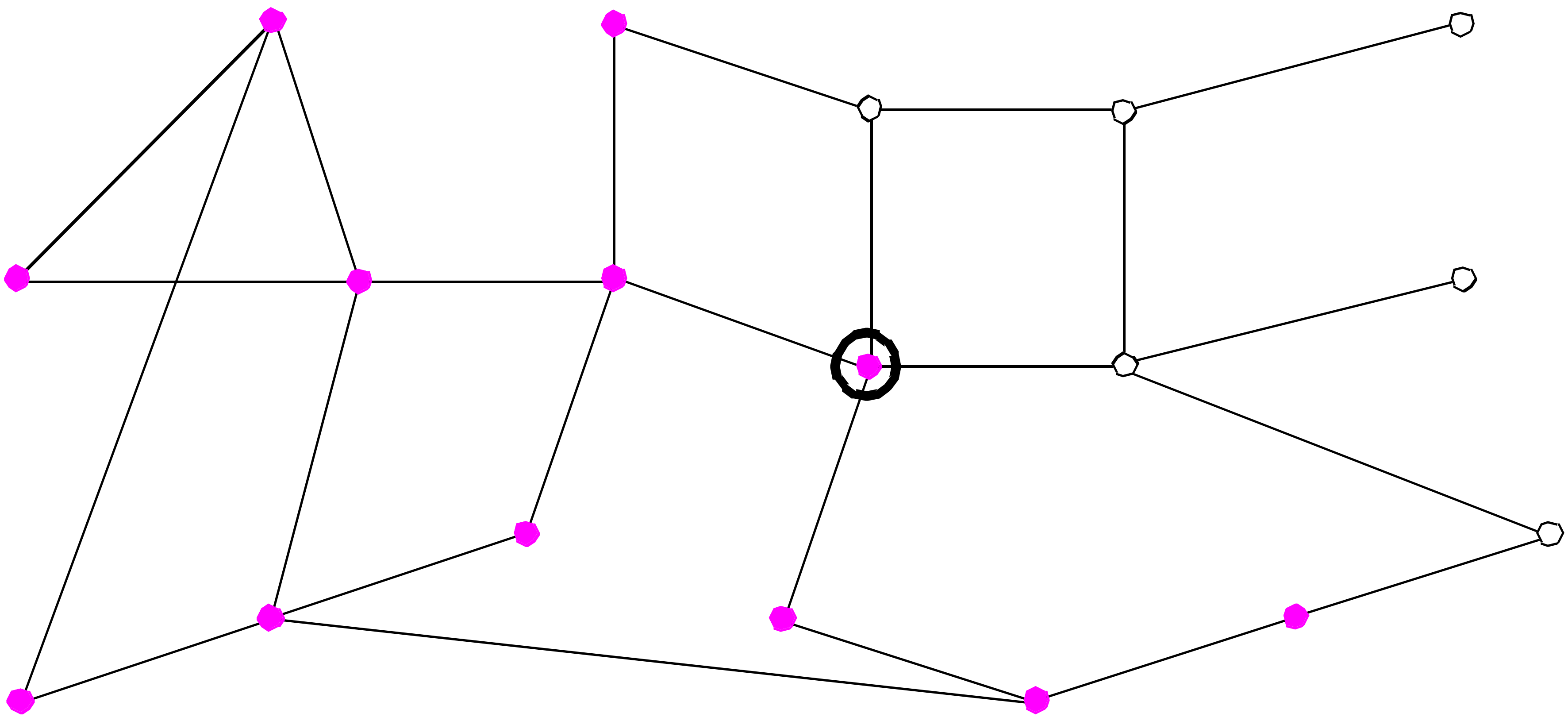
number of movements = 7

Greedy Agent-Centered Search



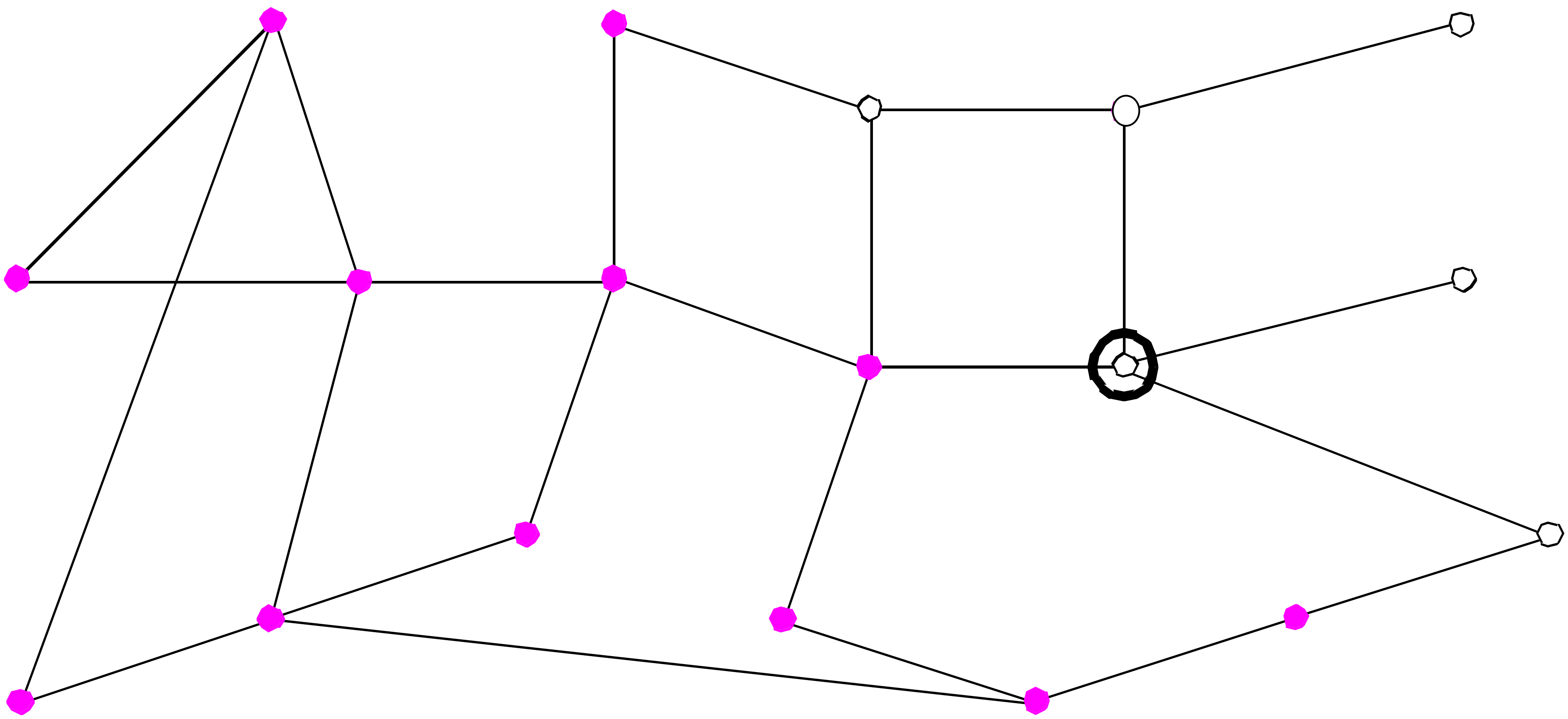
number of movements = 7

Greedy Agent-Centered Search



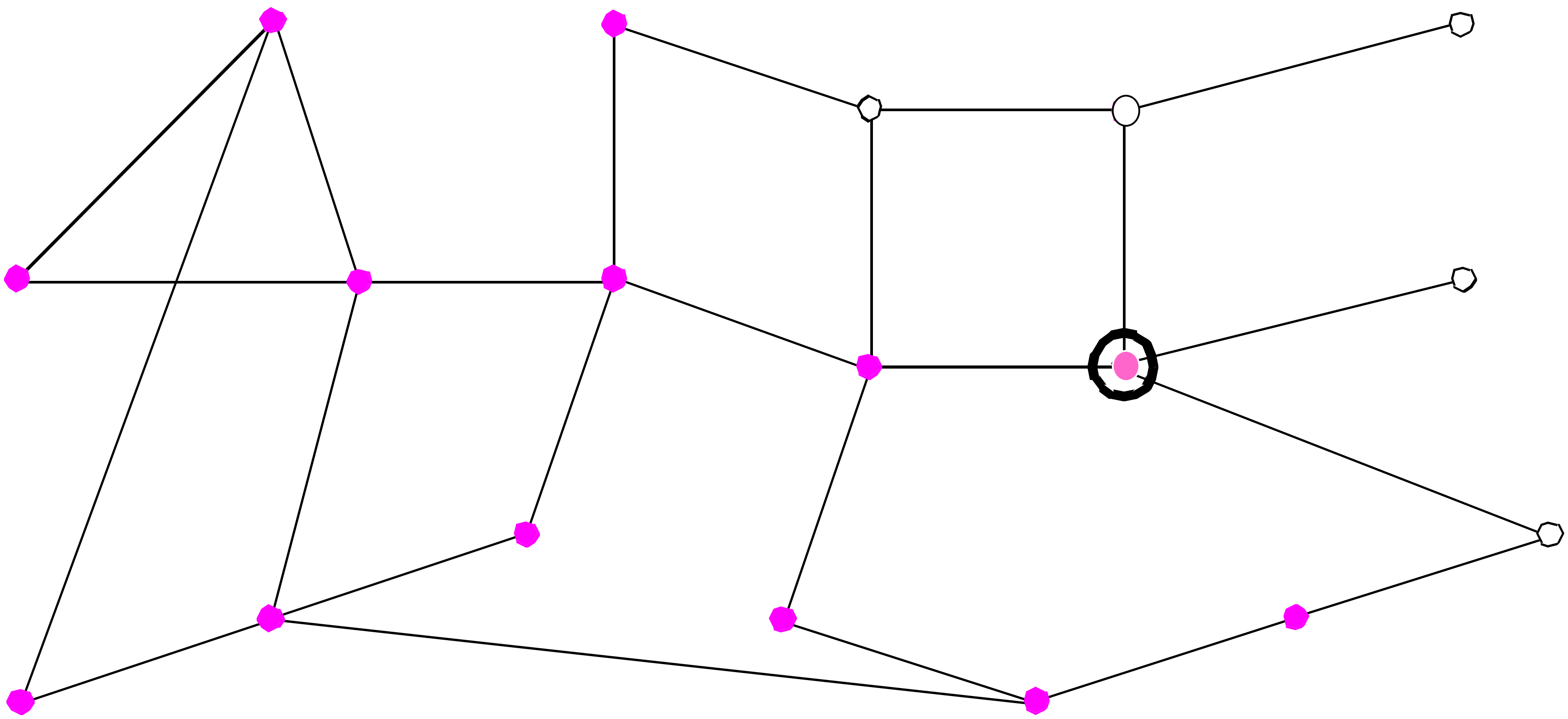
number of movements = 7

Greedy Agent-Centered Search



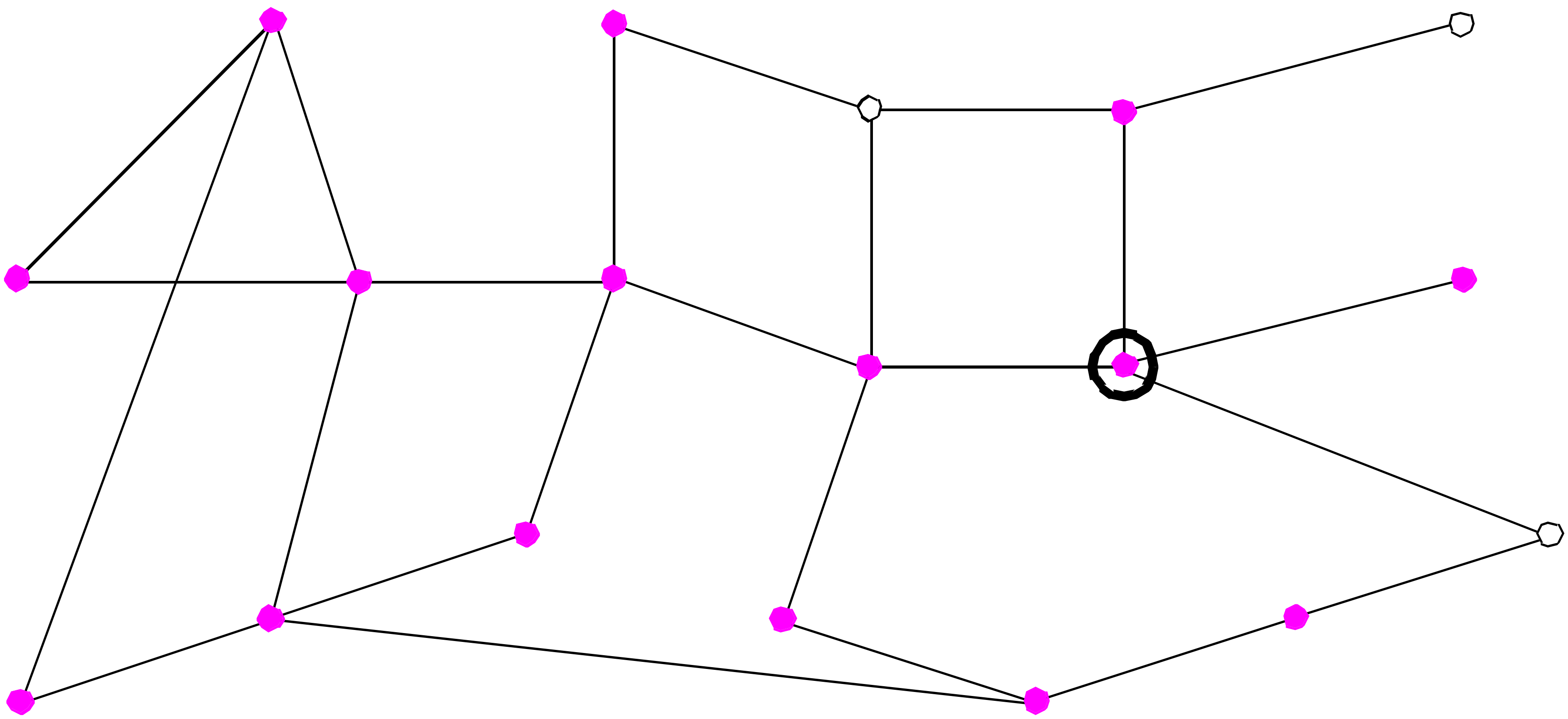
number of movements = 8

Greedy Agent-Centered Search



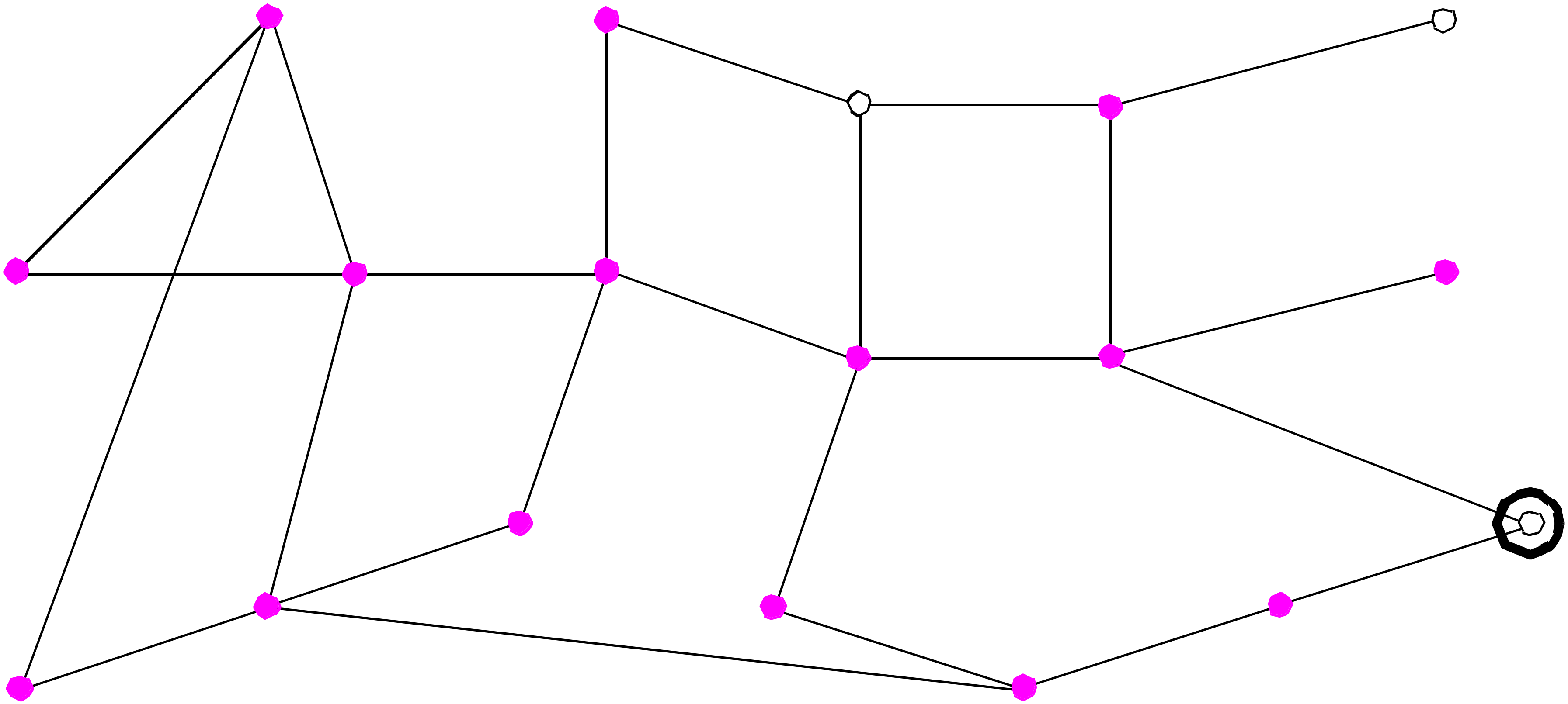
number of movements = 8

Greedy Agent-Centered Search



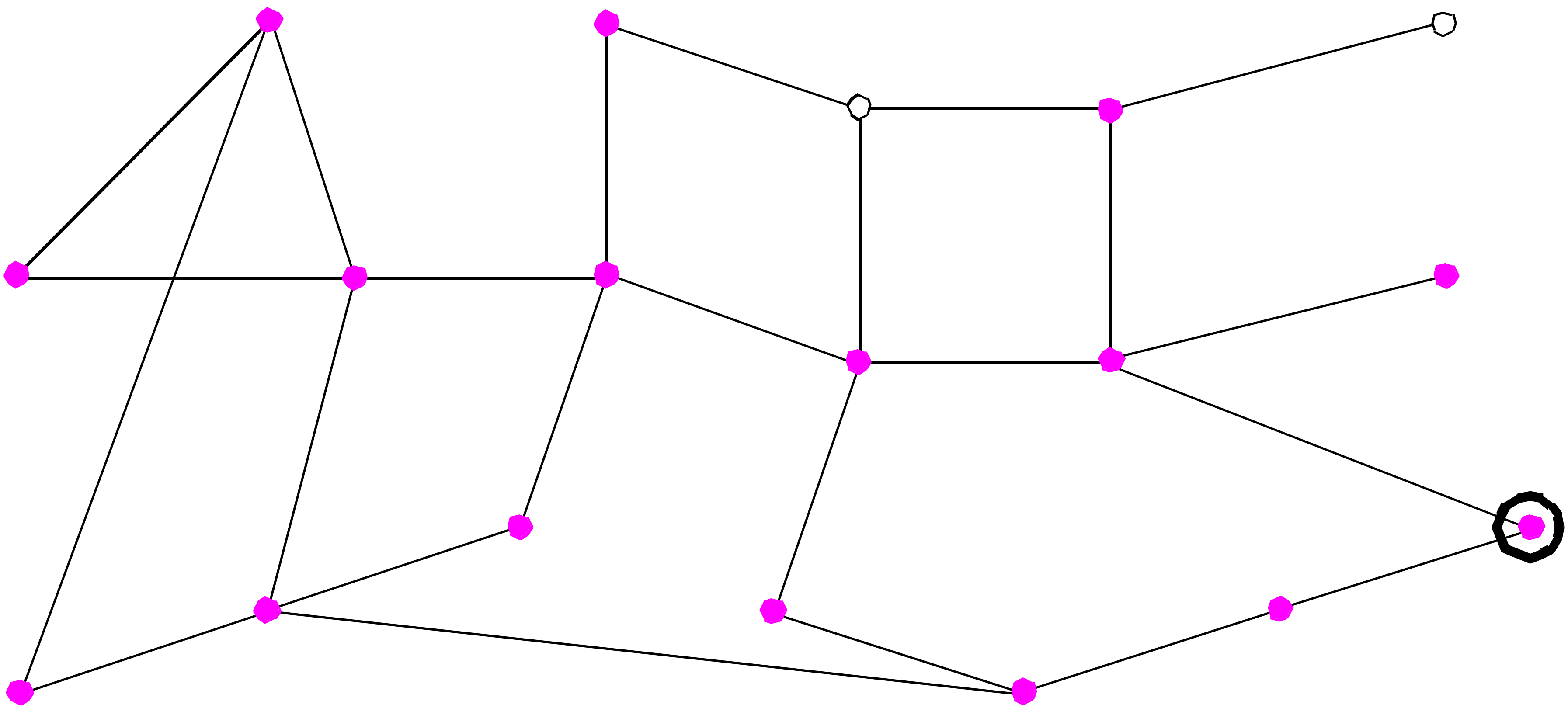
number of movements = 8

Greedy Agent-Centered Search



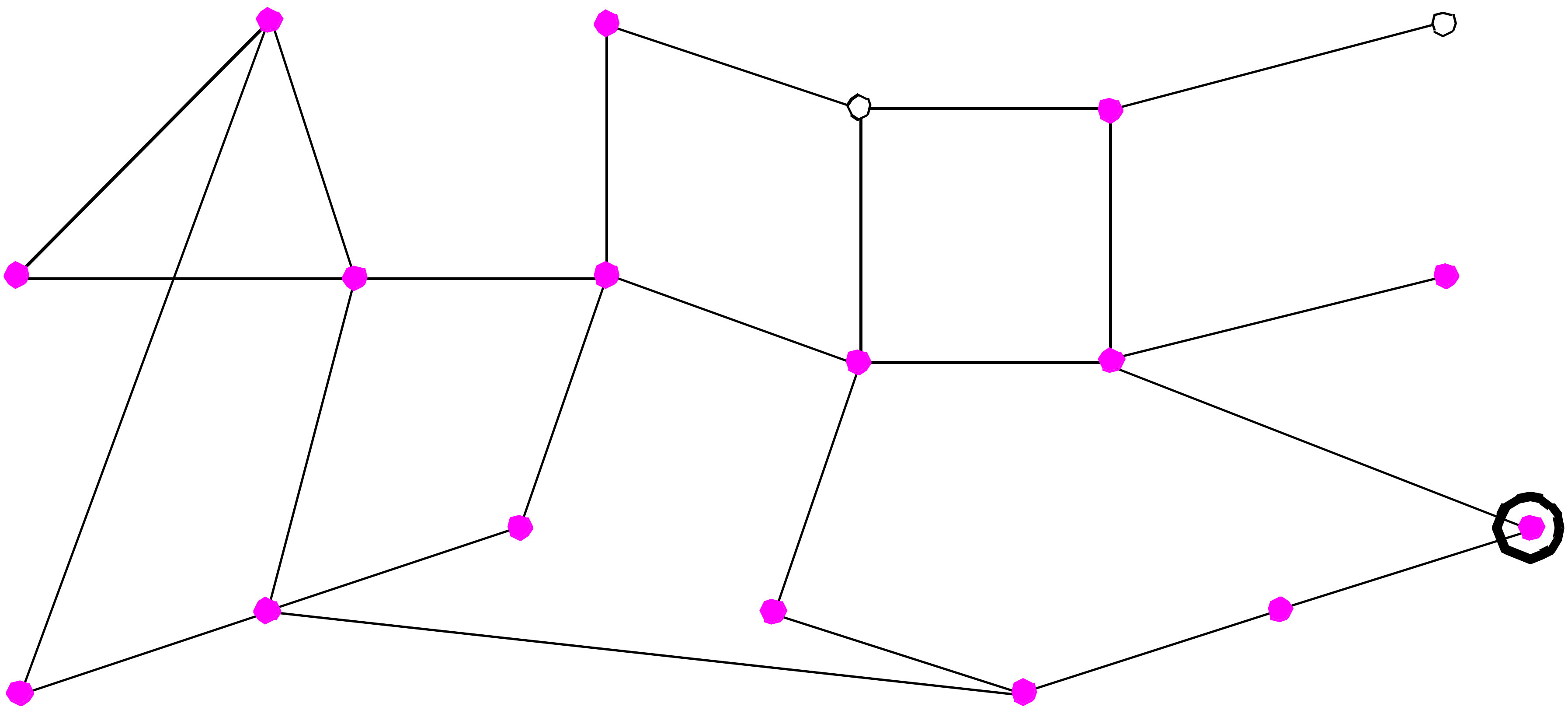
number of movements = 9

Greedy Agent-Centered Search



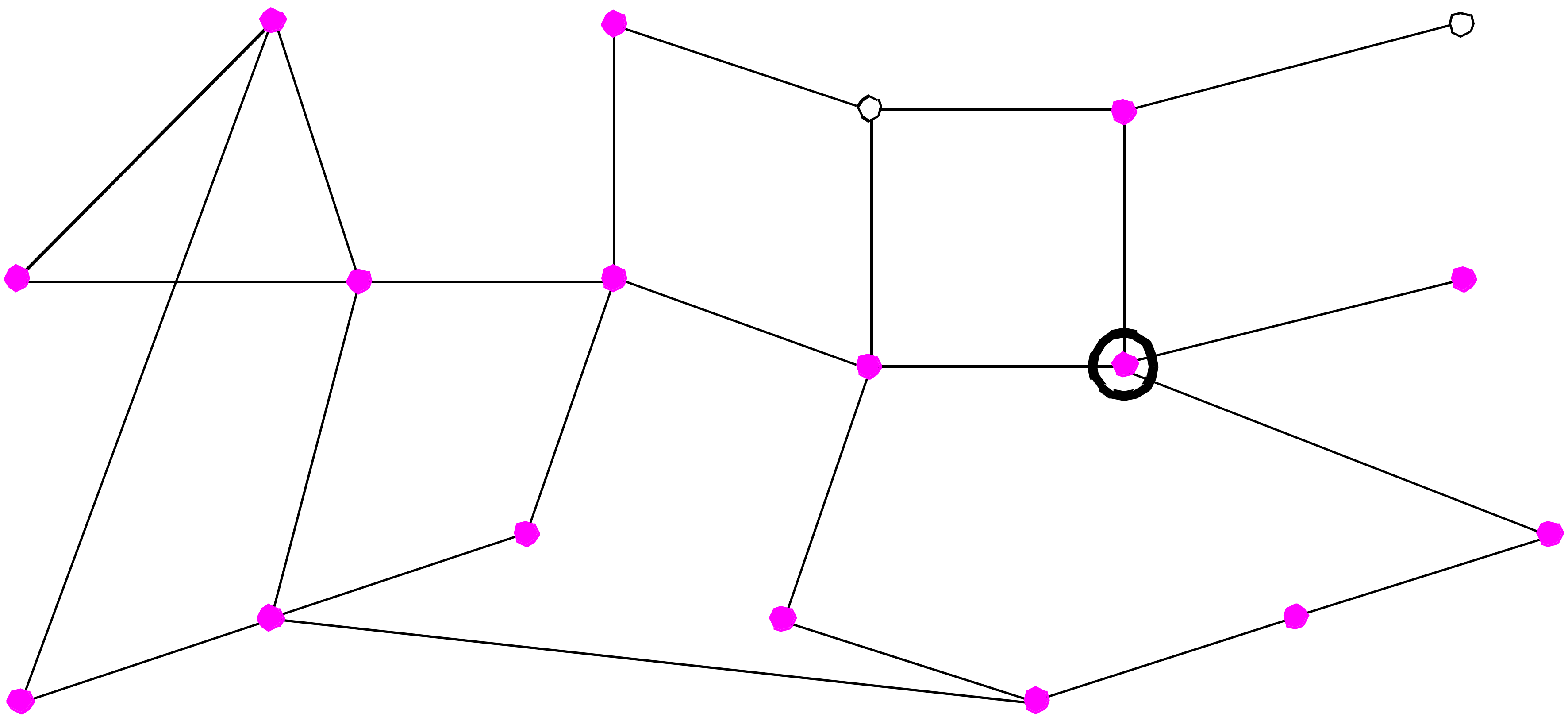
number of movements = 9

Greedy Agent-Centered Search



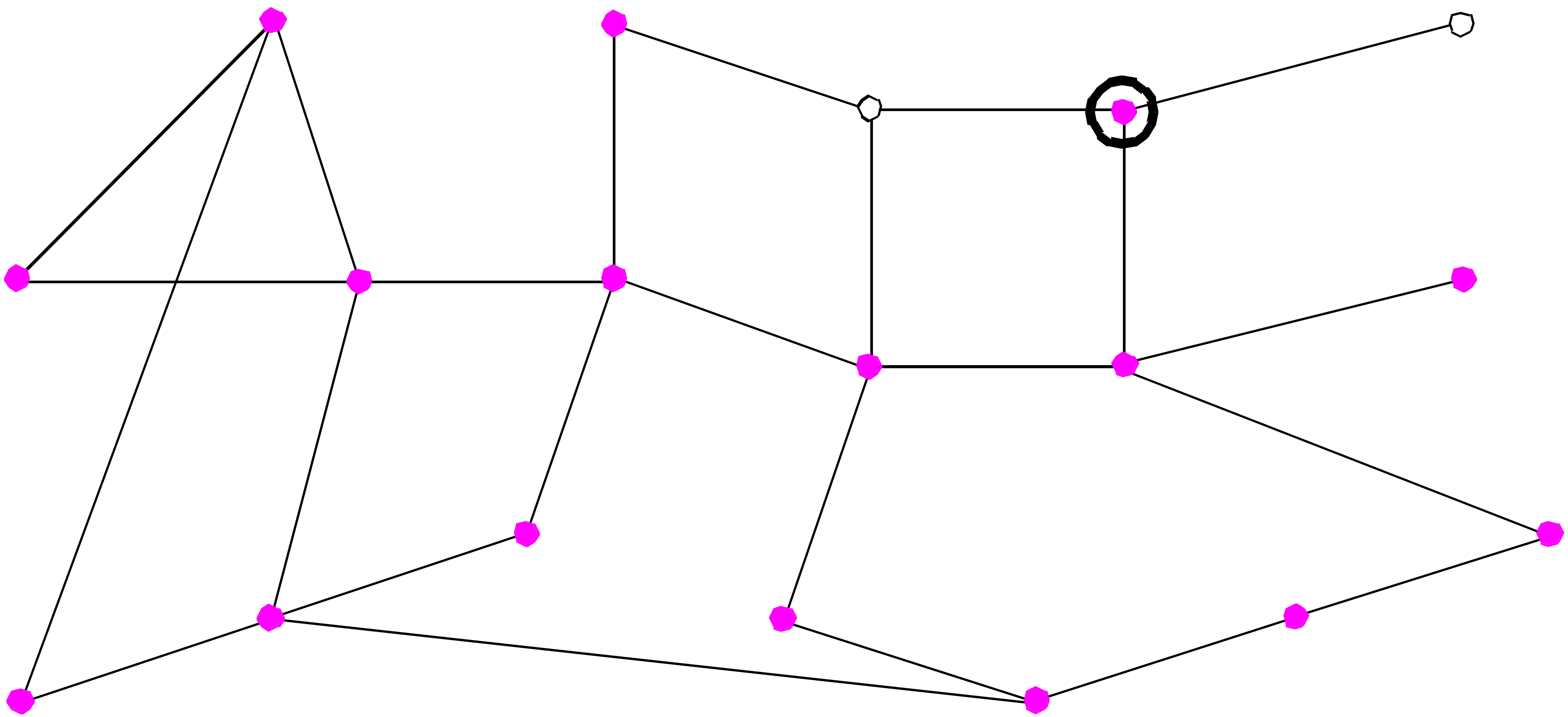
number of movements = 9

Greedy Agent-Centered Search



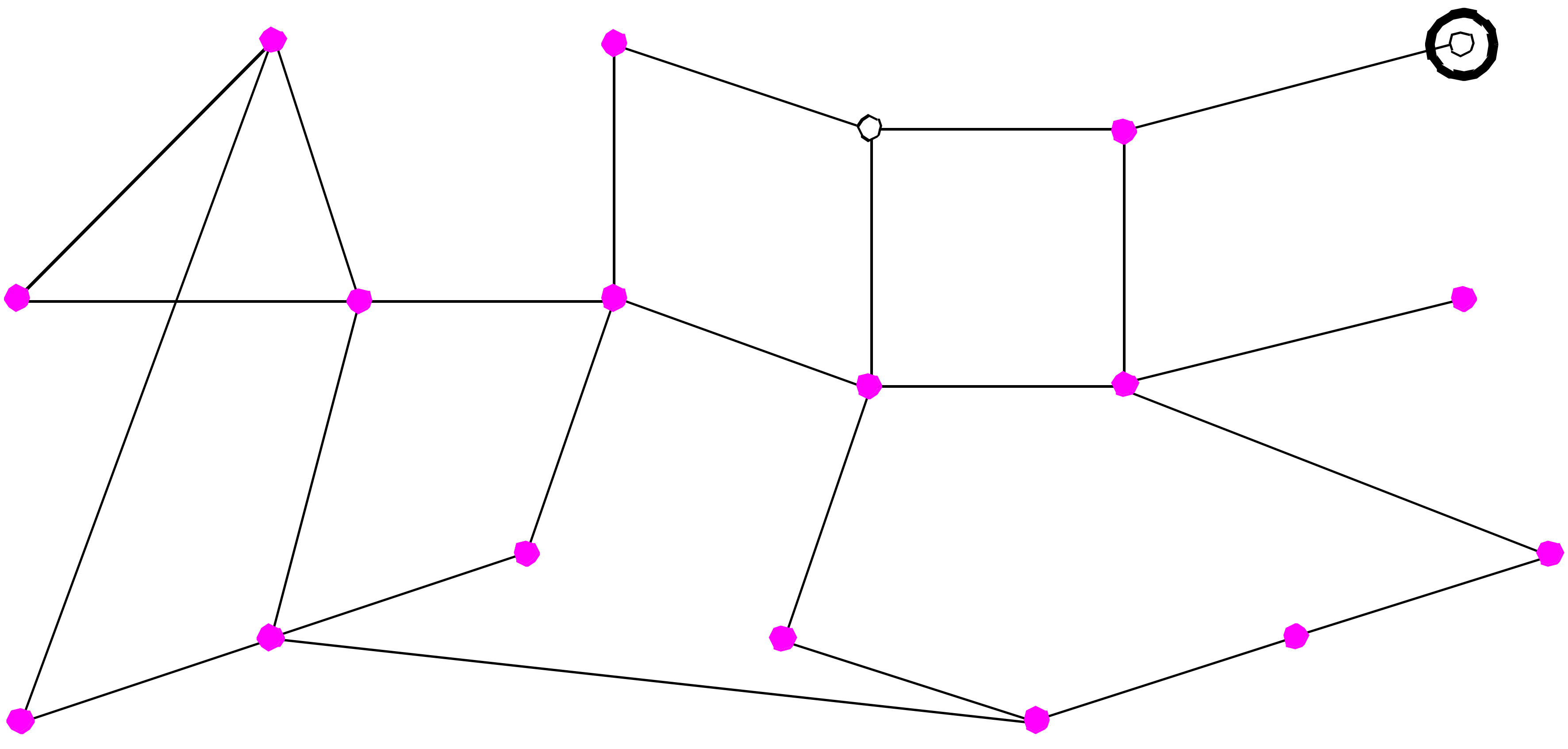
number of movements = 10

Greedy Agent-Centered Search



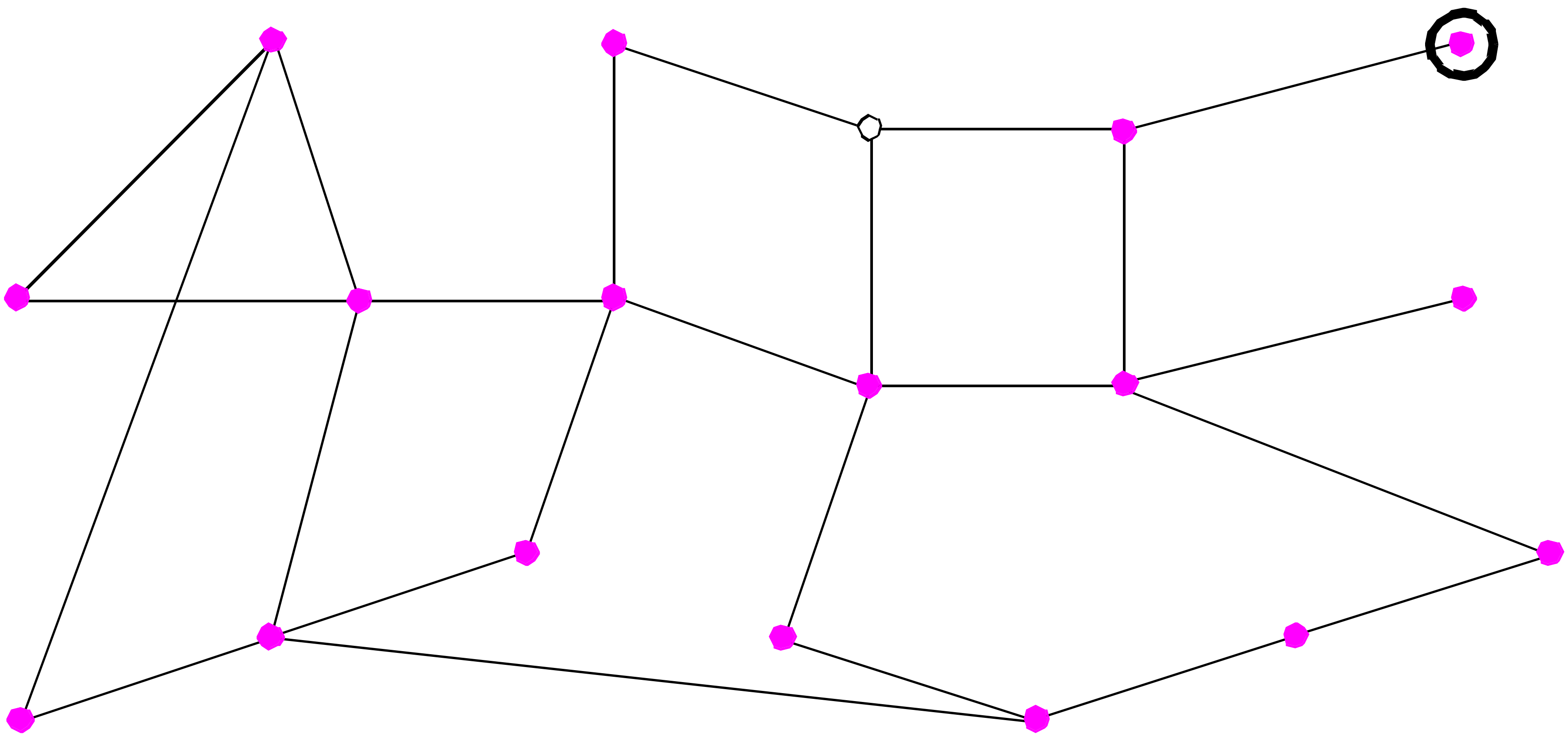
number of movements = 11

Greedy Agent-Centered Search



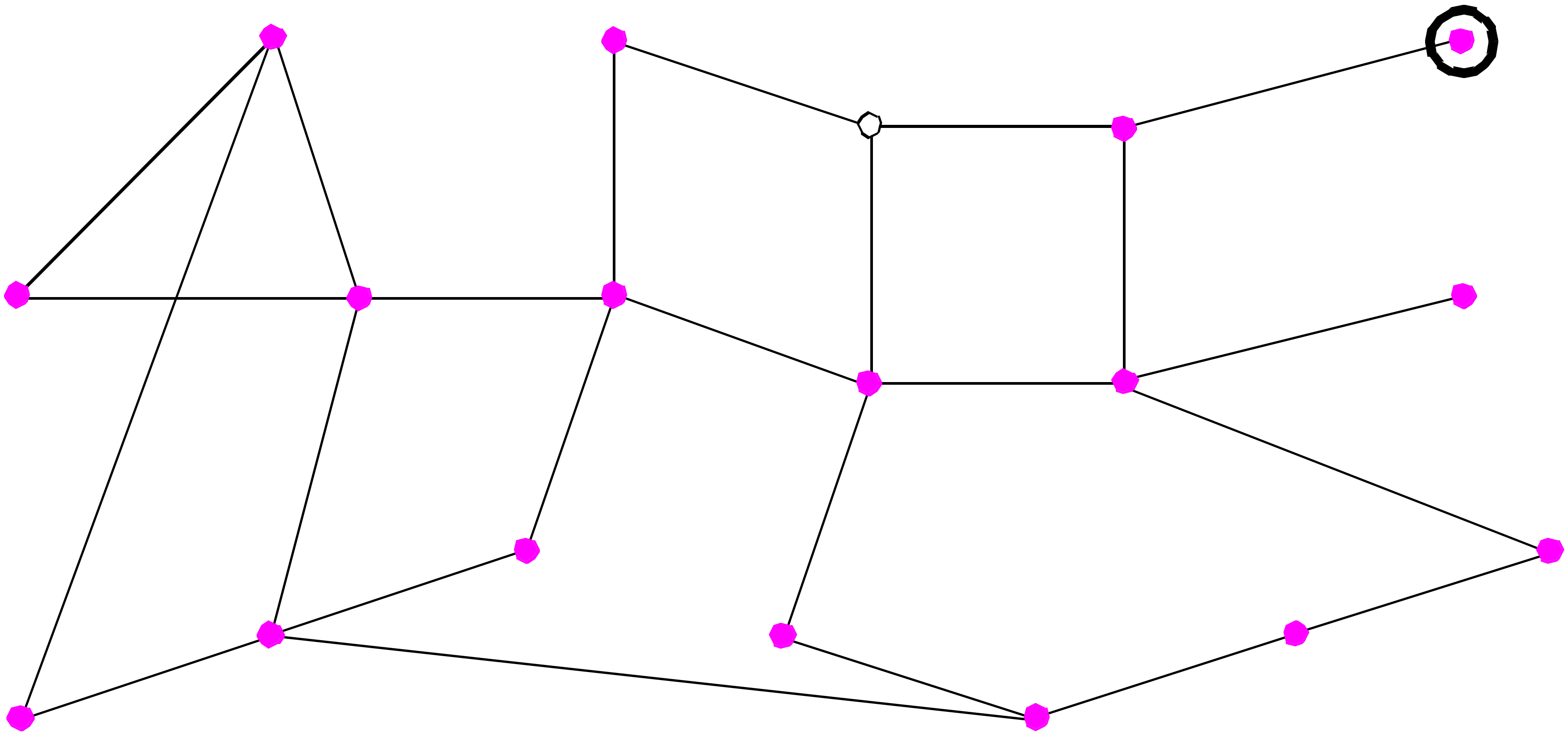
number of movements = 12

Greedy Agent-Centered Search



number of movements = 12

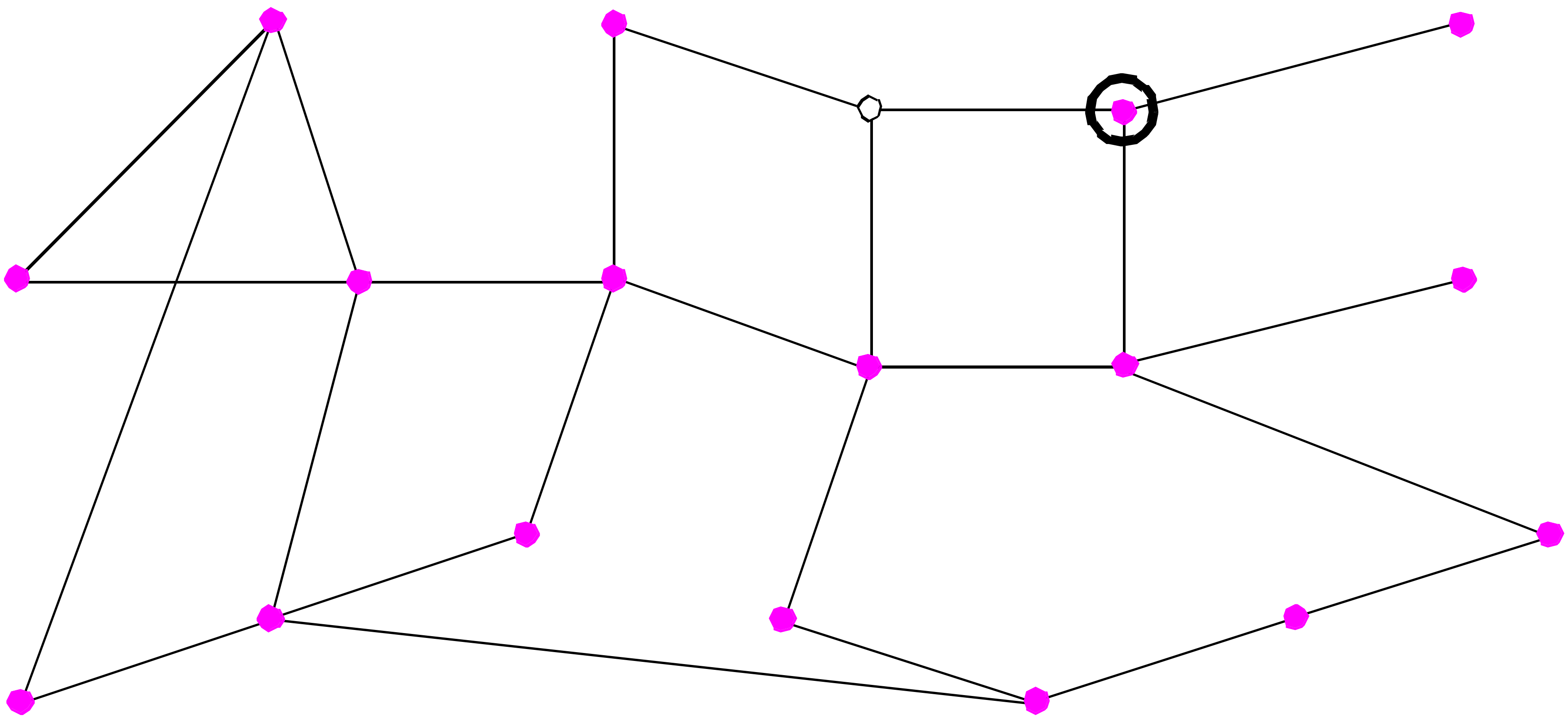
Greedy Agent-Centered Search



number of movements = 12

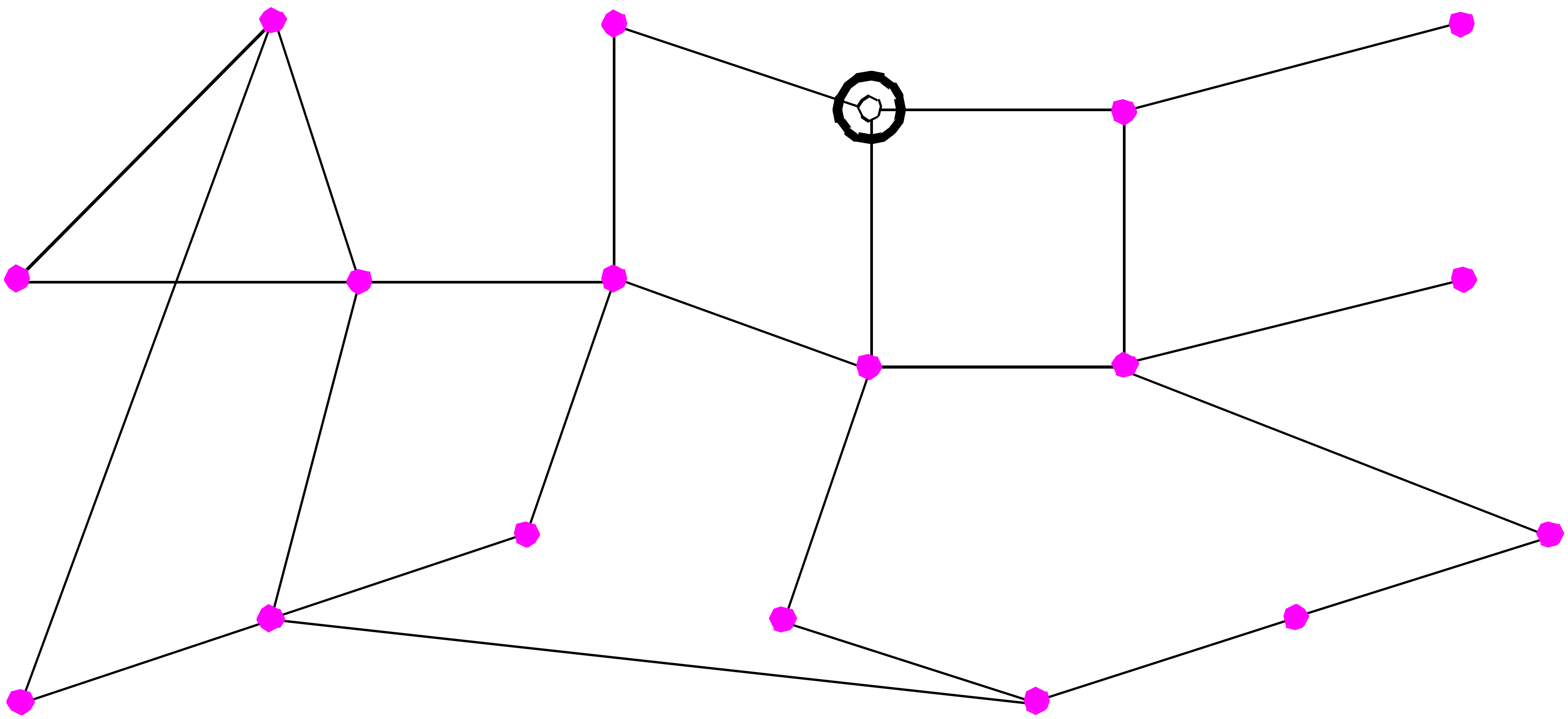
Sven

Greedy Agent-Centered Search



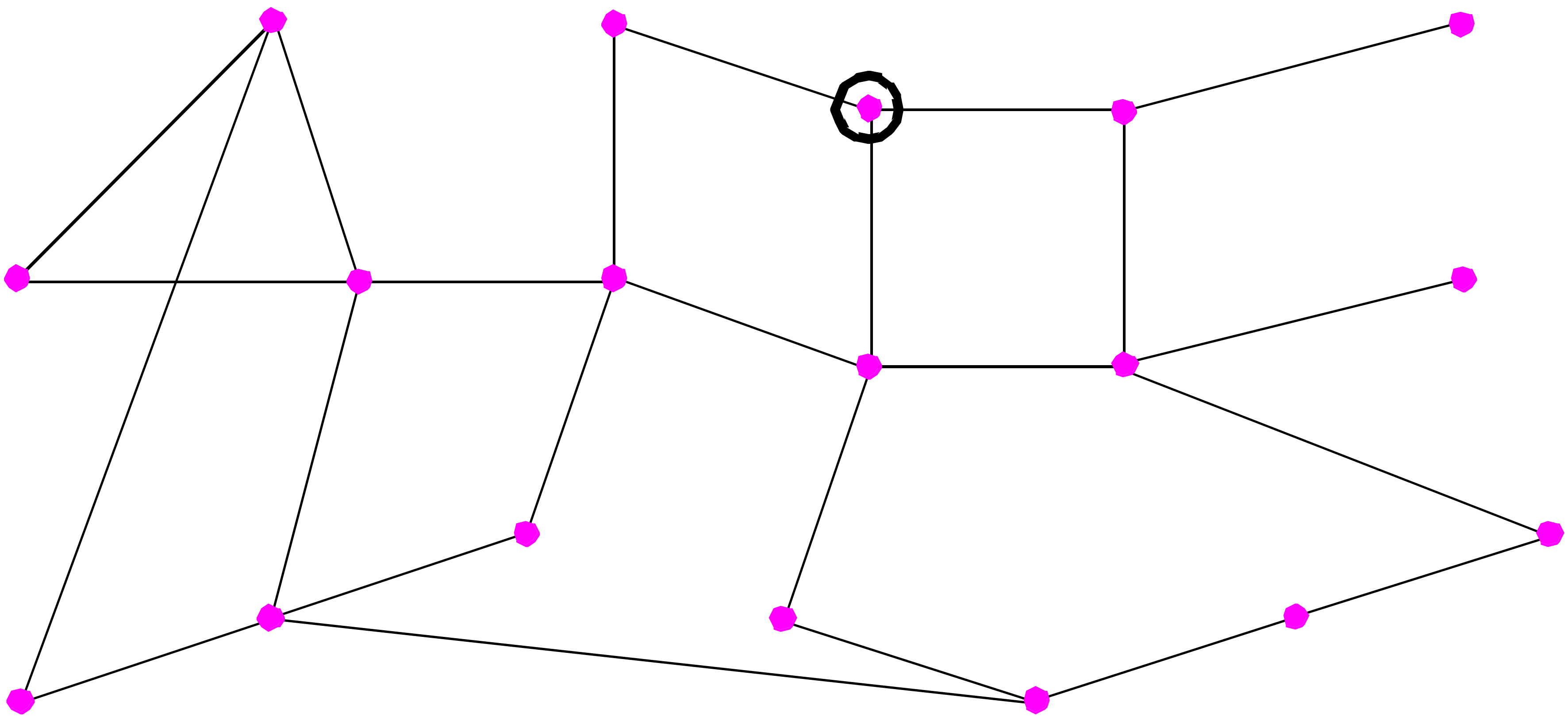
number of movements = 13

Greedy Agent-Centered Search



number of movements = 14

Greedy Agent-Centered Search



number of movements = 14

Greedy Agent-Centered Search

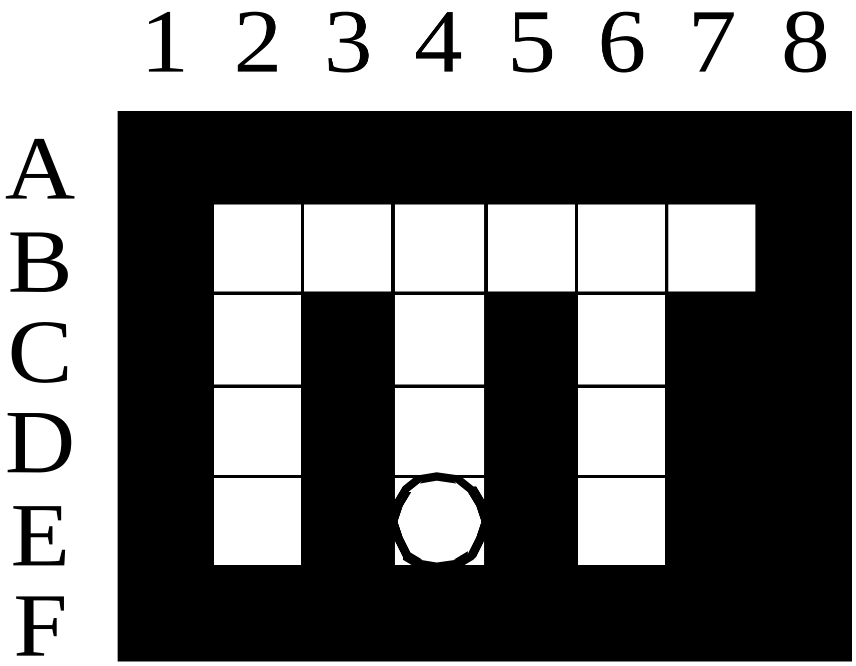
- Theorem [Tovey and Koenig, 2003]

  The worst-case number of movements of greedy agent-centered search is $|V| + 2\,|V| \ln |V|$ in known connected graphs, where $|V|$ is the number of vertices.

Planning Problems and Strategies

- Greedy Agent-Centered Search
- Three Robot-Navigation Problems and Approaches
  - Localization using Agent-Centered Search:
    Greedy Localization
  - Mapping using Agent-Centered Search:
    Greedy Mapping
  - Stationary Target Search in Unknown Terrain
    using Assumption-Based Planning:
    Planning with the Freespace Assumption
- Summary
  - Agent-Centered Search
  - Planning with the Freespace Assumption
  - Real-Time Search

Robot-Navigation Problems



the agent sees

+---

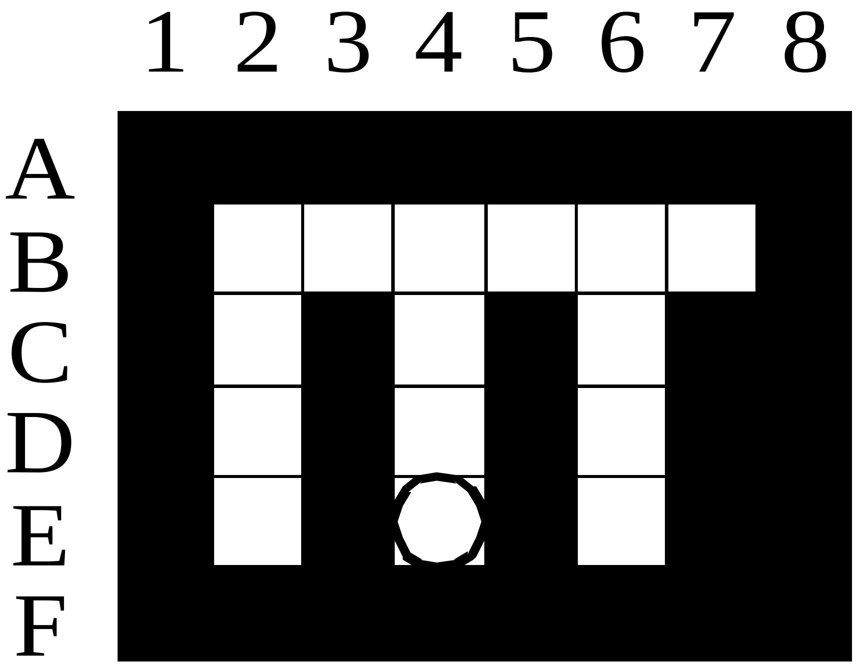
- Perfect actuation in four compass directions
- Perfect sensing in four compass directions with sensor range one
- Compass is available
- Minimize the worst-case number of movements for
  - Grid of a given size
  - Start cell
  - Tie breaking

- Greedy Agent-Centered Search
- Three Robot-Navigation Problems and Approaches
  - Localization using Agent-Centered Search:
    Greedy Localization
  - Mapping using Agent-Centered Search:
    Greedy Mapping
  - Stationary Target Search in Unknown Terrain
    using Assumption-Based Planning:
    Planning with the Freespace Assumption
- Summary
  - Agent-Centered Search
  - Planning with the Freespace Assumption
  - Real-Time Search

Localization

- Localization determines the robot cell on a given map.

Localization

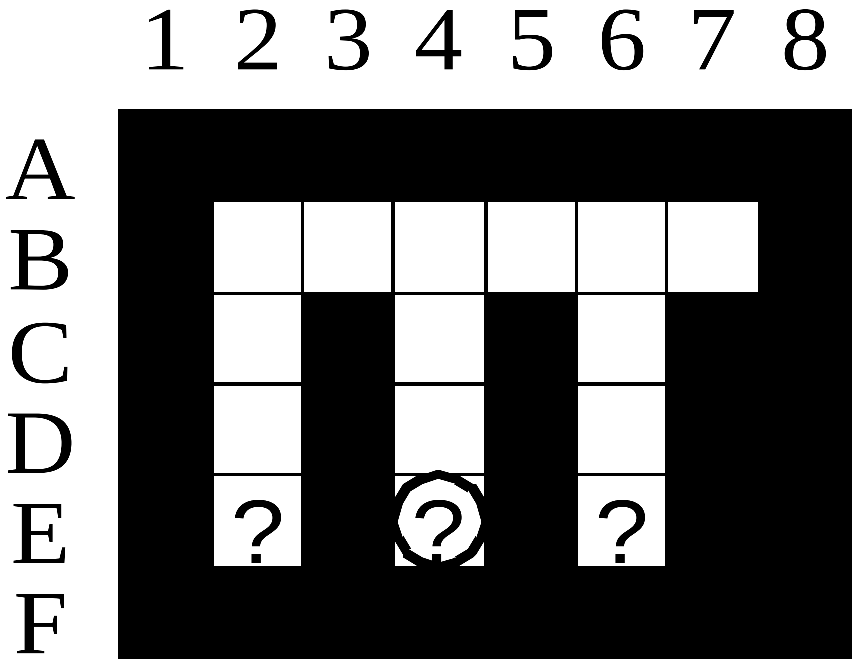


the agent sees

+---

4-neighbor grid

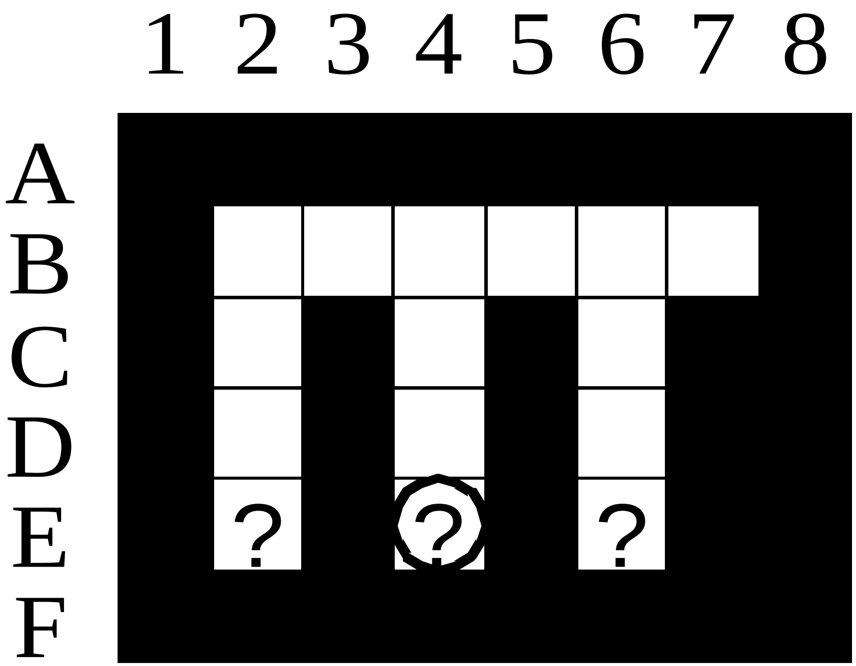
Localization



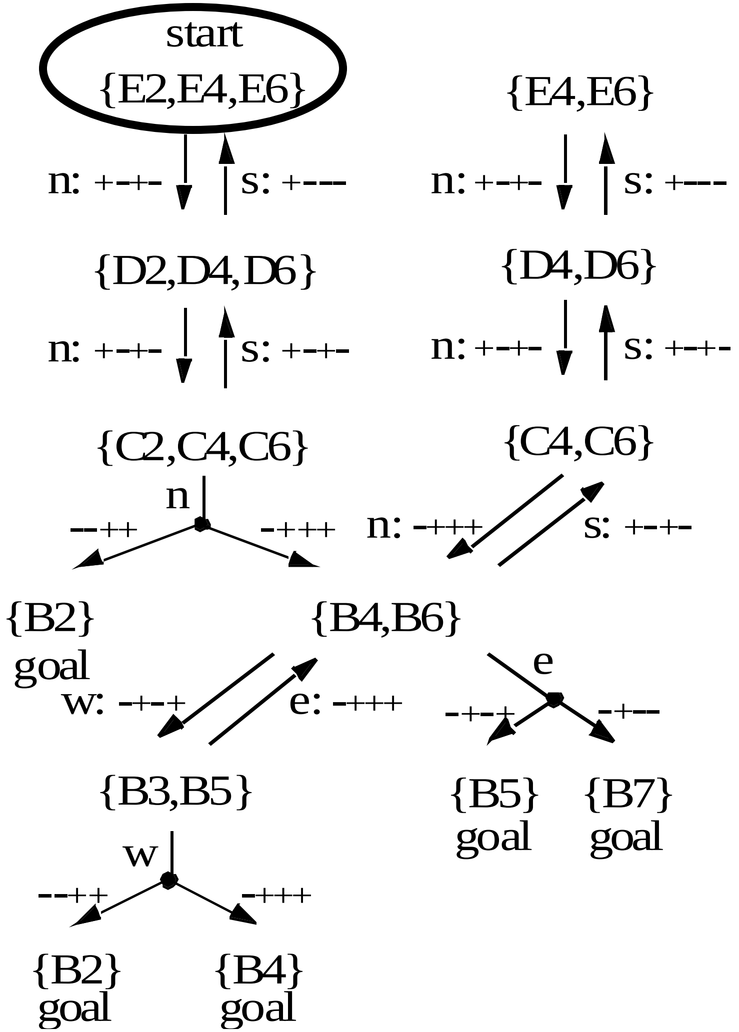
the agent could be in
{E2, E4, E6}

4-neighbor grid

Localization

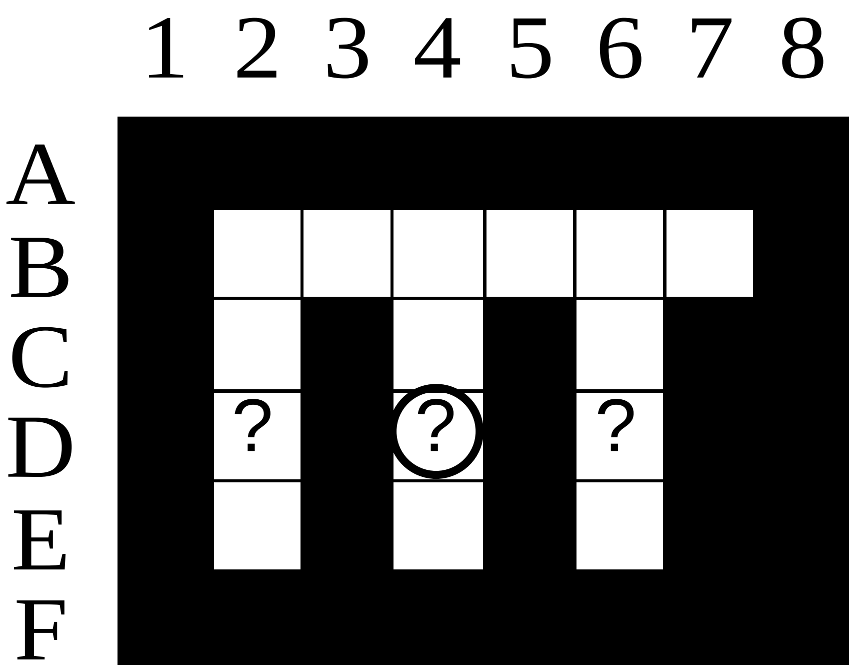


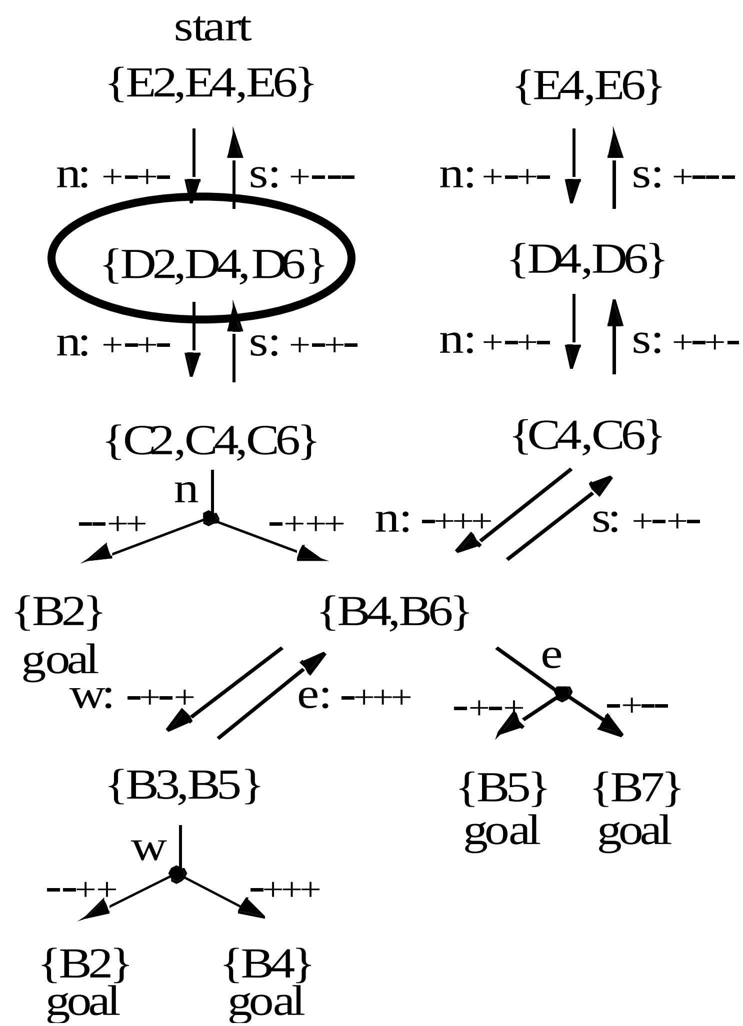
the agent could be in
{E2, E4, E6}



4-neighbor grid

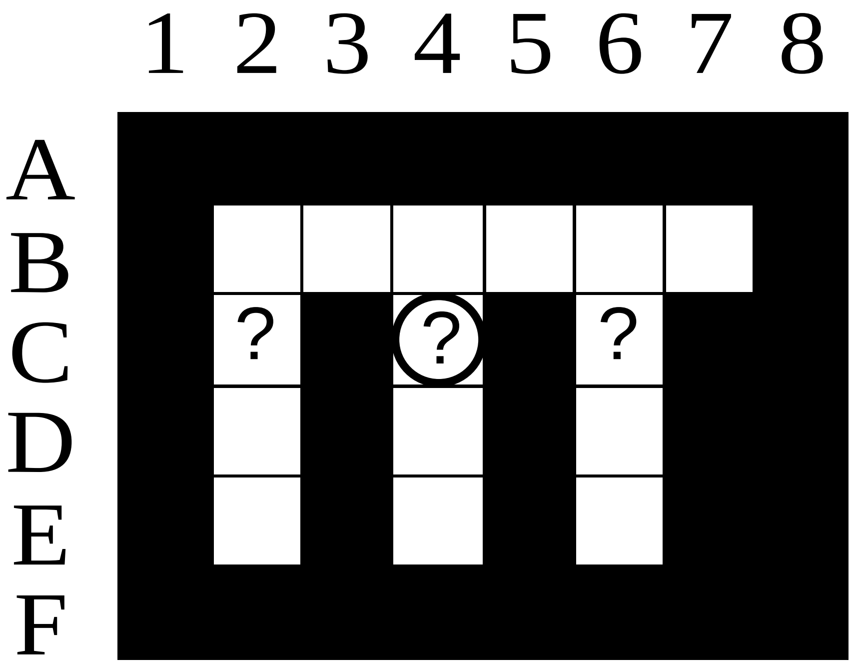
Localization

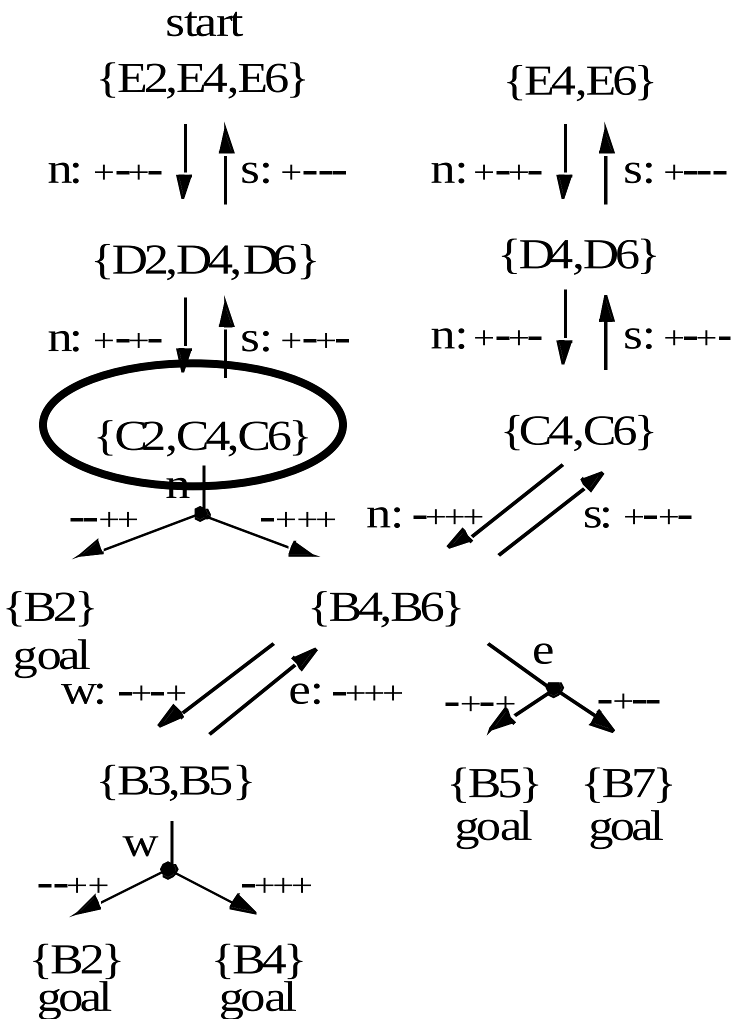




4-neighbor grid

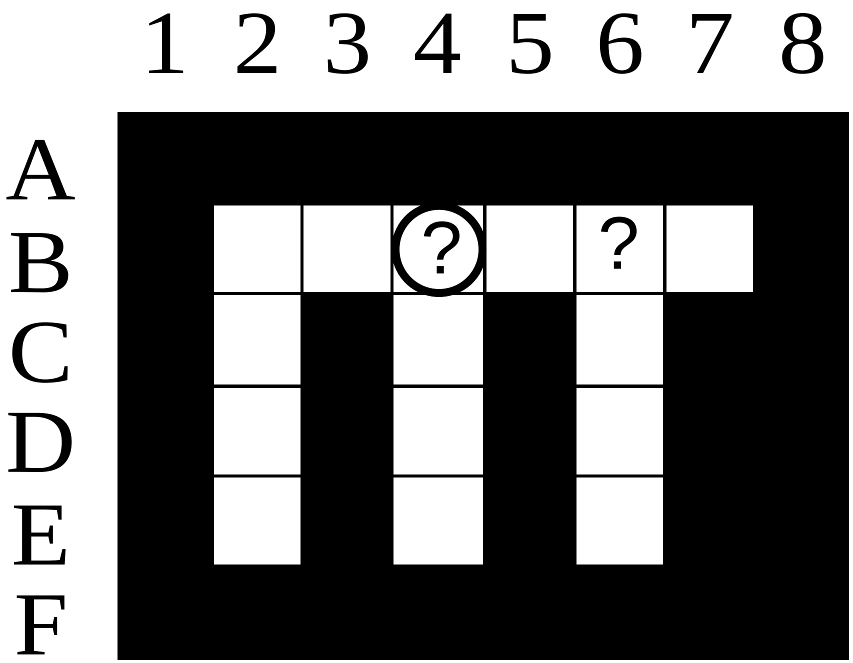
Localization

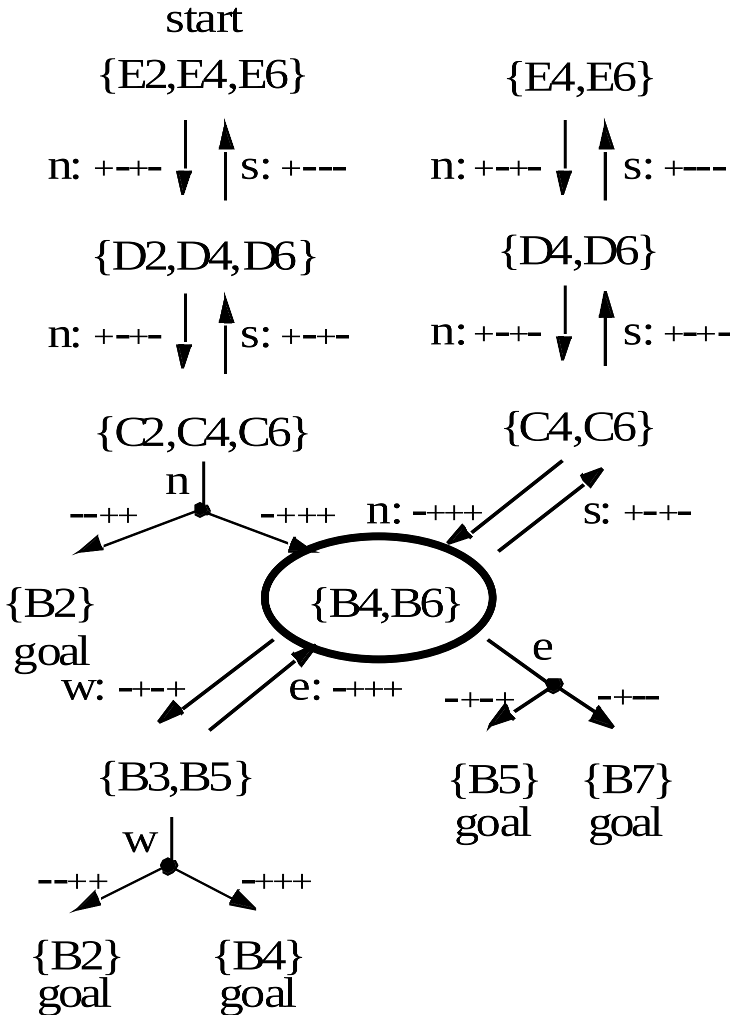




4-neighbor grid

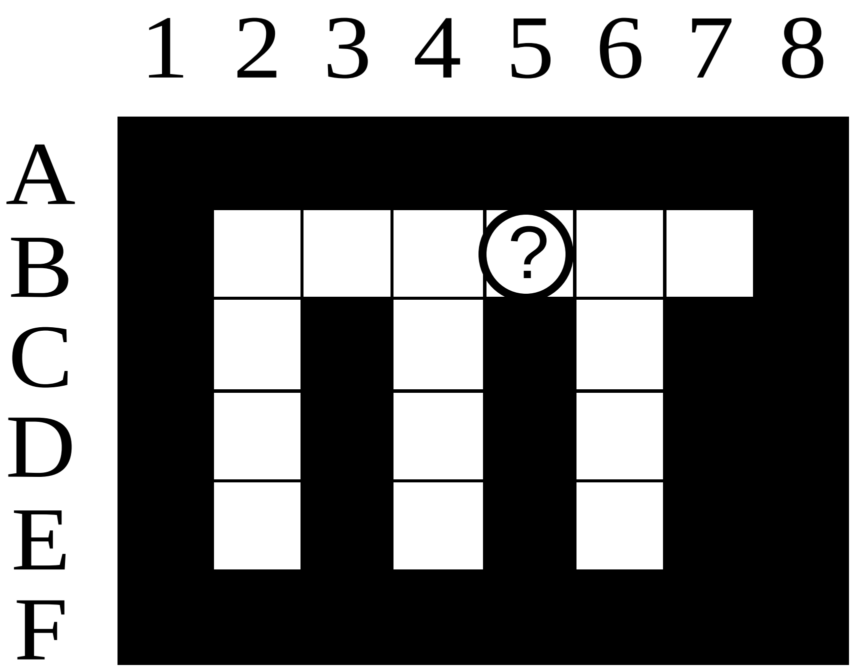
Localization

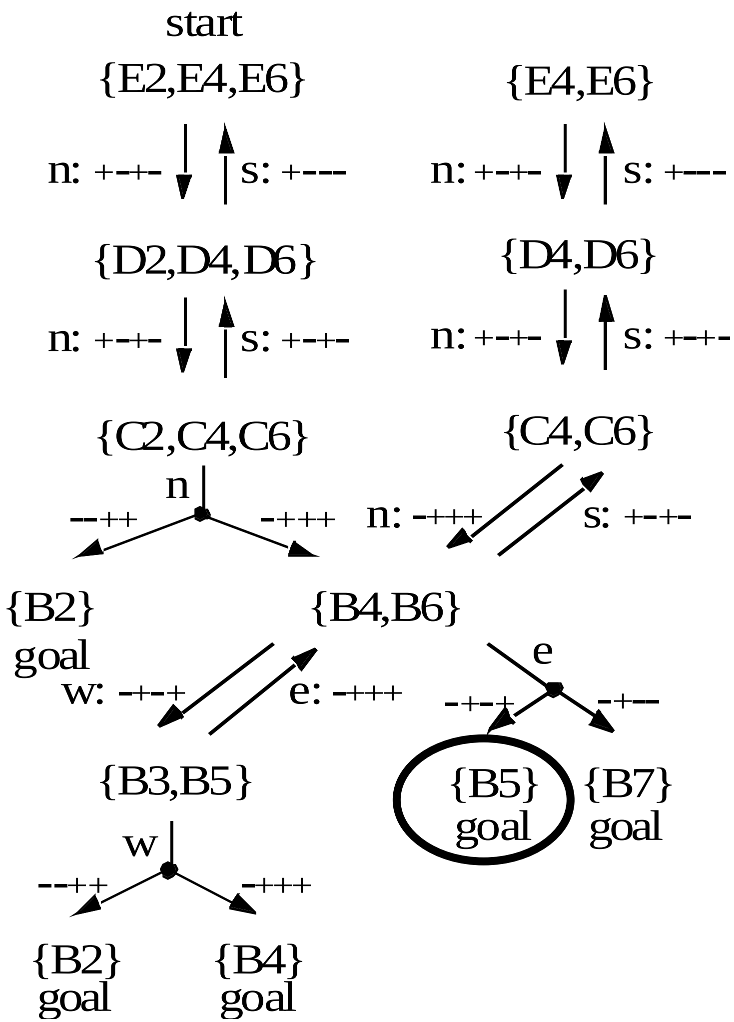




4-neighbor grid

Localization





4-neighbor grid

Approx Optimal Localization

- Theorem [Tovey and Koenig, 2000]

  It is in NP to determine whether there exists a localization plan that executes no more movements than a given value.

  It is NP-hard to find a localization plan in grids whose worst-case number of movements to localization is within a factor O(log(|V|)) of optimum, where |V| is the number of states (= unblocked cells), even in connected grids in which localization is possible.
  (Contrast this theorem with [Dudek, Romanik, Whitesides, 1995].)

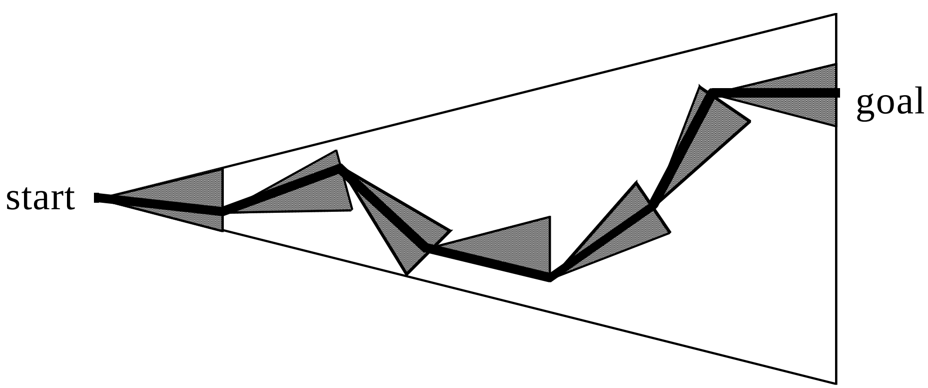
- Thus, it is intractable to find optimal localization plans via complete AND-OR searches.

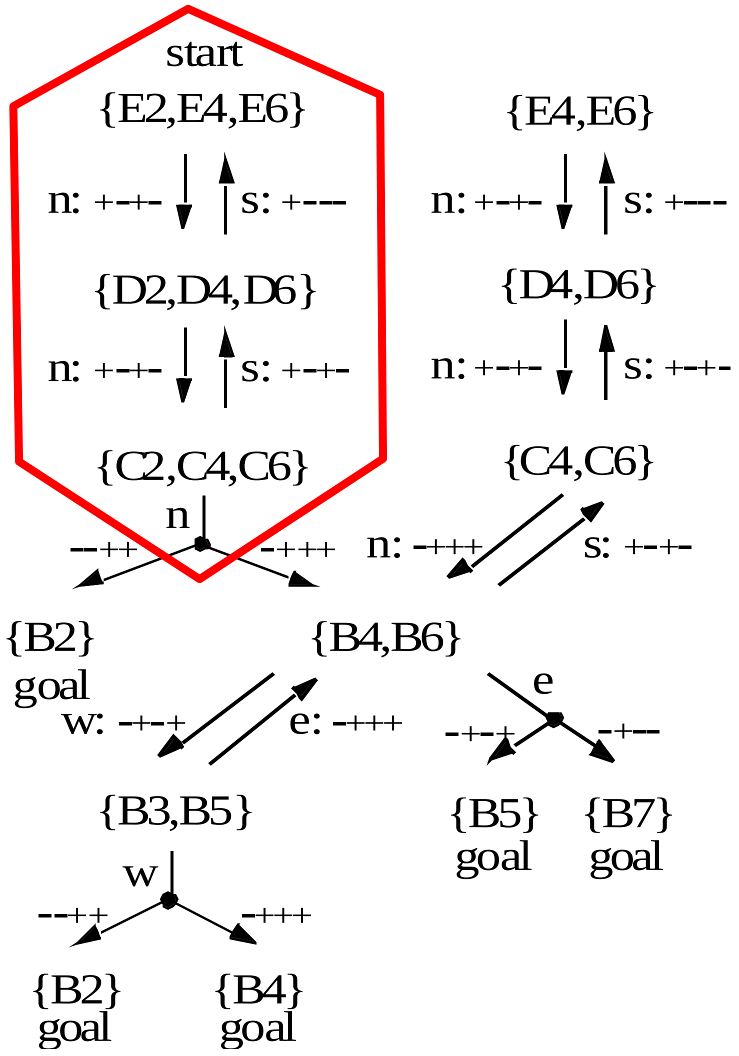
Approx Optimal Localization

	Approx optimal localization
Planning time	Exponential
Plan-execution time	Low-order polynomial

Greedy Localization

- Agent-centered search: interleaving of deterministic searches that result in a gain in information with action executions.

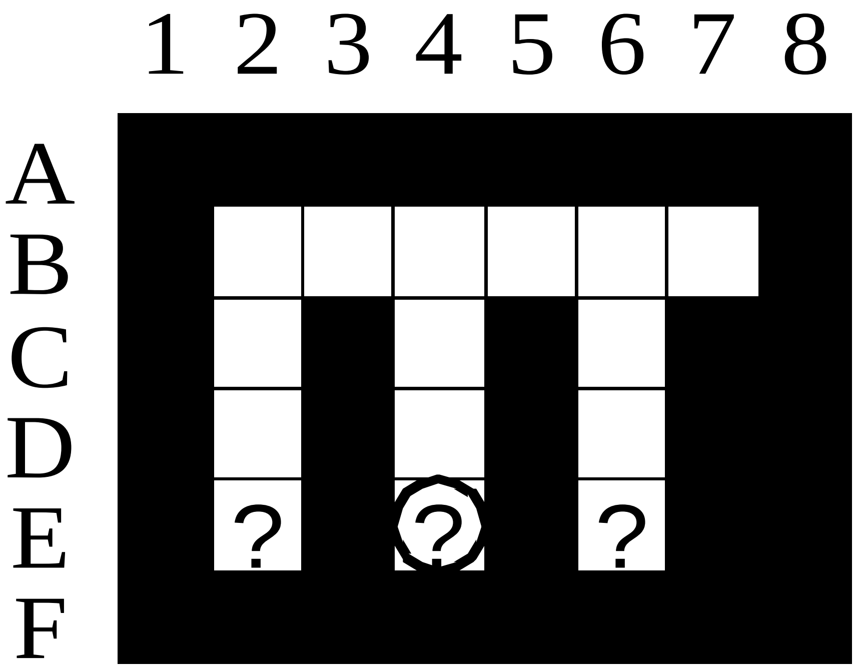


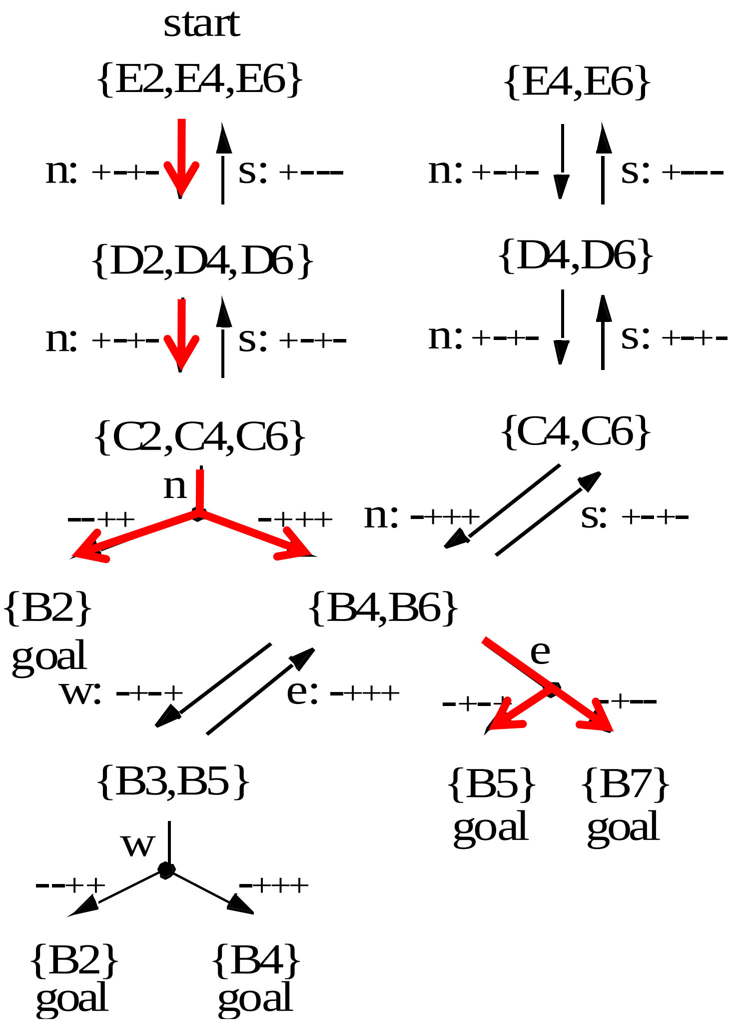


Greedy Localization

- Greedy localization repeatedly makes the robot execute a shortest movement sequence to a closest informative unblocked cell, where an informative cell is one that allows the robot to make an observation that is guaranteed to reduce the number of possible robot cells [Genesereth and Nourbakhsh, 1993] [Koenig and Simmons, 1998].

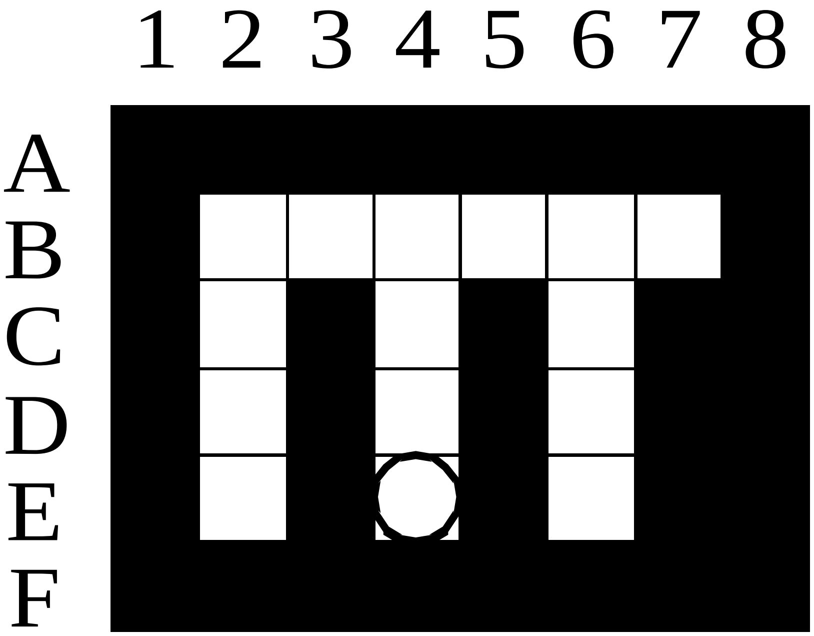
Greedy Localization

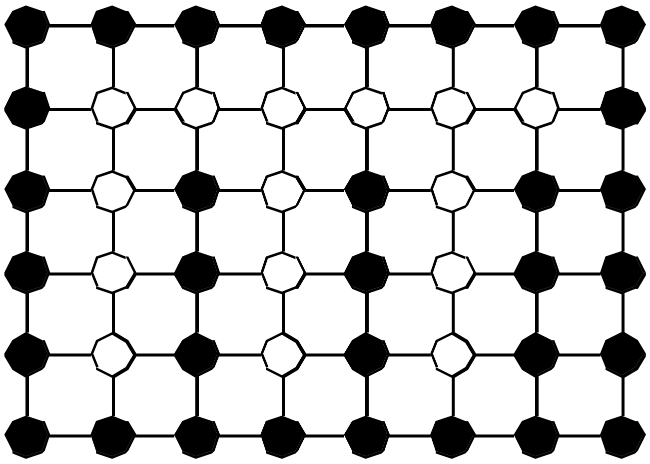




4-neighbor grid

Greedy Localization





4-neighbor grid

Greedy Localization

- Greedy localization starts at some unblocked cell. It marks the robot cell (and perhaps other cells as well) as uninformative and then moves to the closest informative unblocked cell. It repeats the process until all unblocked cells are marked uninformative.

- Corollary [Tovey and Koenig, 2005]

  The worst-case number of movements of greedy localization is O(|V| log |V|), where |V| is the number of states (= unblocked cells).

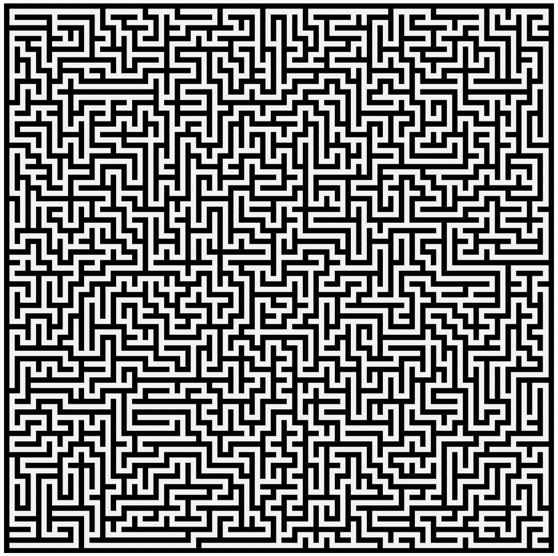
Greedy Localization

	Approx optimal localization
Planning time	Exponential
Plan-execution time	Low-order polynomial

	Greedy Localization
Planning time	Low-order polynomial
Plan-execution time	Low-order polynomial

Greedy Localization

- DFS mazes



Acyclic mazes generated with DFS

Greedy Localization

- DFS mazes

gridworld size	obstacle density	av. number of subplans to localization		av. number of steps per subplan to localization		av. total number of movements to localization
11 x 11	41.3 %	2.4	x	1.5	=	3.6
21 x 21	45.4 %	3.3	x	1.7	=	5.4
31 x 31	46.8 %	3.8	x	1.7	=	6.6
41 x 41	47.6 %	4.1	x	1.8	=	7.5
51 x 51	48.1 %	4.5	x	1.8	=	8.0
61 x 61	48.4 %	4.7	x	1.8	=	8.6
71 x 71	48.6 %	4.9	x	1.9	=	9.1

(5041 cells)

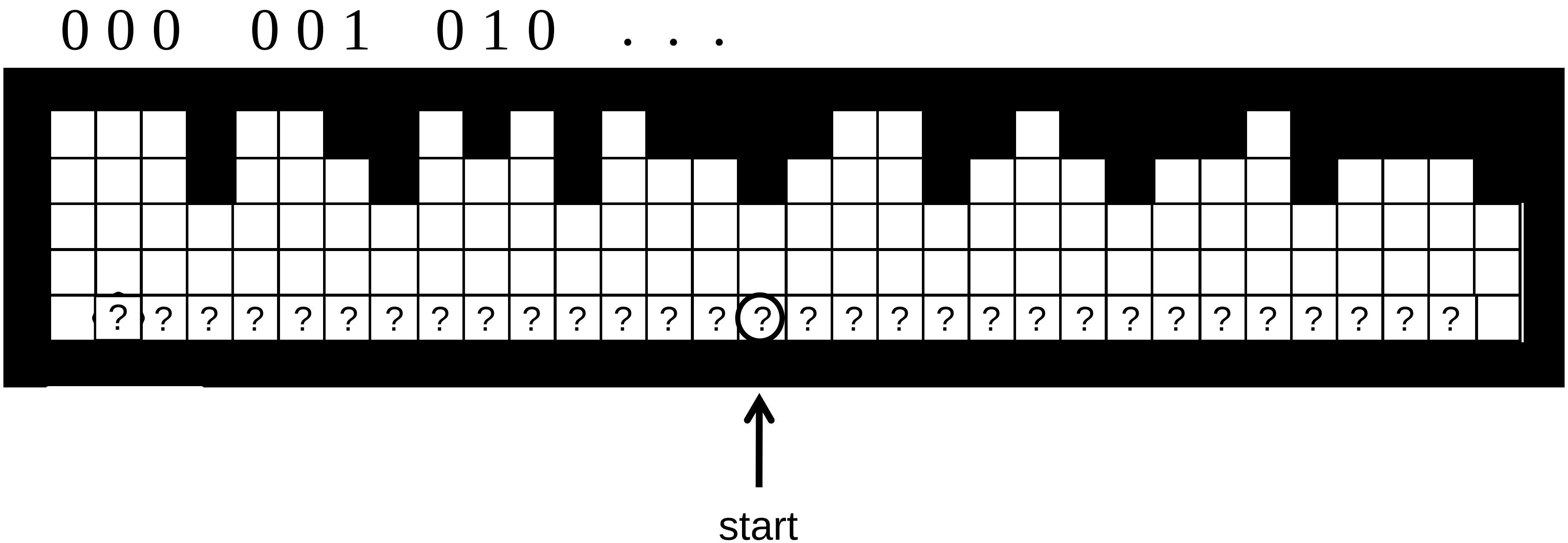
Greedy Localization

- Example for room-like terrain [Tovey and Koenig, 2005]

  The worst-case number of movements of greedy localization can be a factor $\Omega(|V| / \log |V|)$ worse than the optimal worst-case number of movements to localization, where $|V|$ is the number of states (= unblocked cells), even in connected grids in which localization is possible.

Greedy Localization

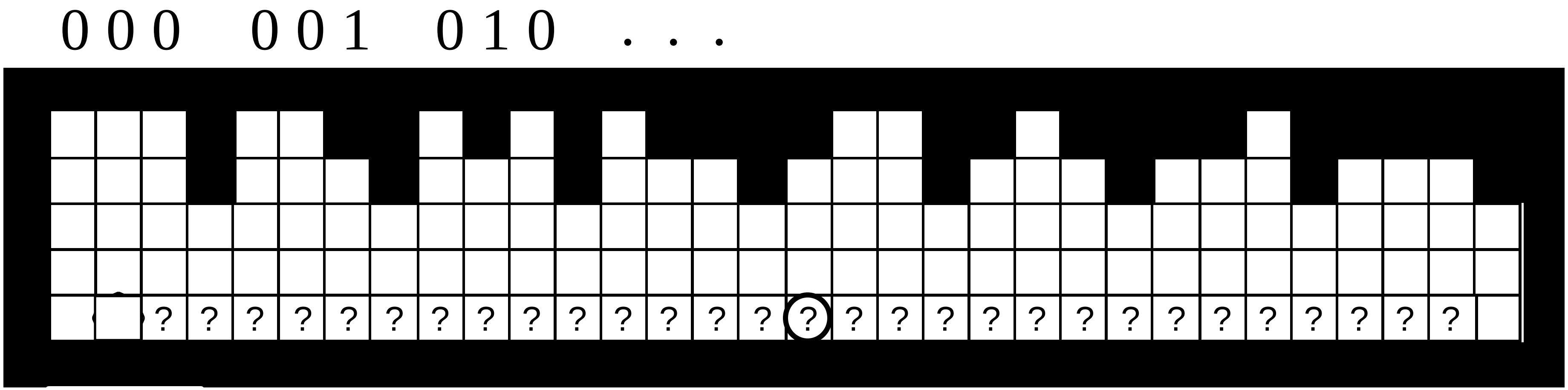
- Our grids



4-neighbor grid

Greedy Localization

- Our grids



Greedy Localization

	Approx optimal localization
Planning time	Exponential
Plan-execution time	Low-order polynomial

	Greedy Localization
Planning time	Low-order polynomial
Plan-execution time	Low-order polynomial

Greedy Localization

- Our minimax model
  - Perfect actuation, perfect sensing
  - Minimize worst-case number of movements
  - Sets of states

- POMDP-based ("Markov") localization [Burgard, Fox and Thrun, 1997]
  - Noisy actuation, noisy sensing
  - Minimize average-case number of movements
  - Probability distribution over states

Greedy Localization

- Our minimax model
  - Greedy localization repeatedly makes the robot execute a shortest movement sequence that is guaranteed to reduce the number of possible robot cells.
- POMDP-based ("Markov") localization [Burgard, Fox and Thrun, 1997]
  - Greedy localization repeatedly makes the robot execute a shortest movement sequence that is guaranteed to reduce the entropy of the probability distribution over the possible robot cells.

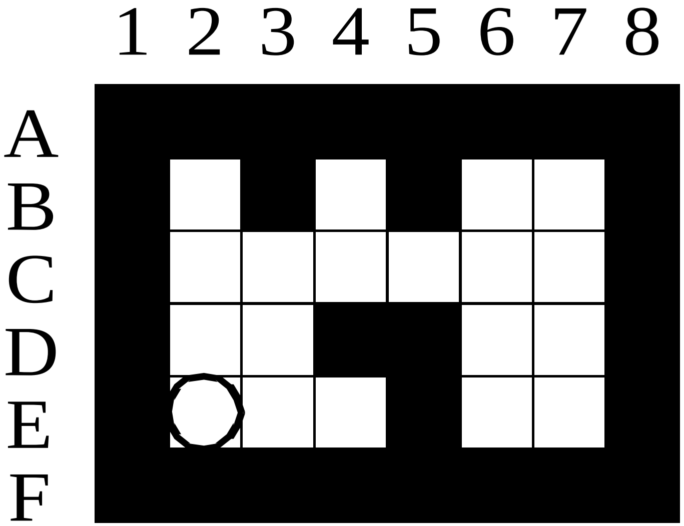
Planning Problems and Strategies

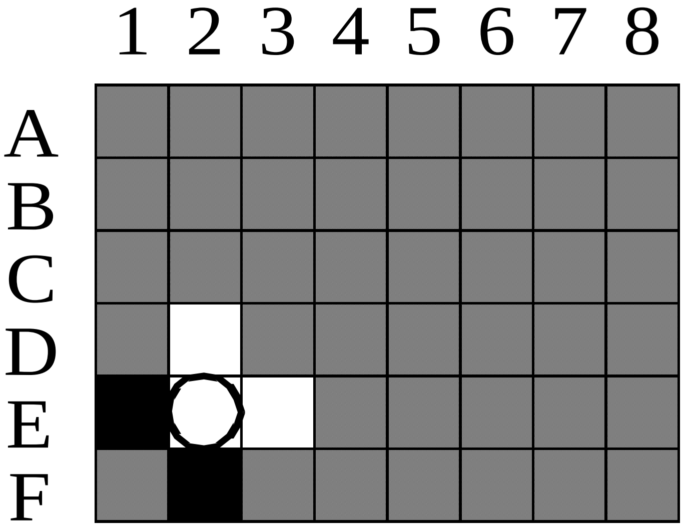
- Greedy Agent-Centered Search
- Three Robot-Navigation Problems and Approaches
  - Localization using Agent-Centered Search:
    Greedy Localization
  - Mapping using Agent-Centered Search:
    Greedy Mapping
  - Stationary Target Search in Unknown Terrain
    using Assumption-Based Planning:
    Planning with the Freespace Assumption
- Summary
  - Agent-Centered Search
  - Planning with the Freespace Assumption
  - Real-Time Search

Mapping

- Mapping determines a map, always knowing the robot cell.

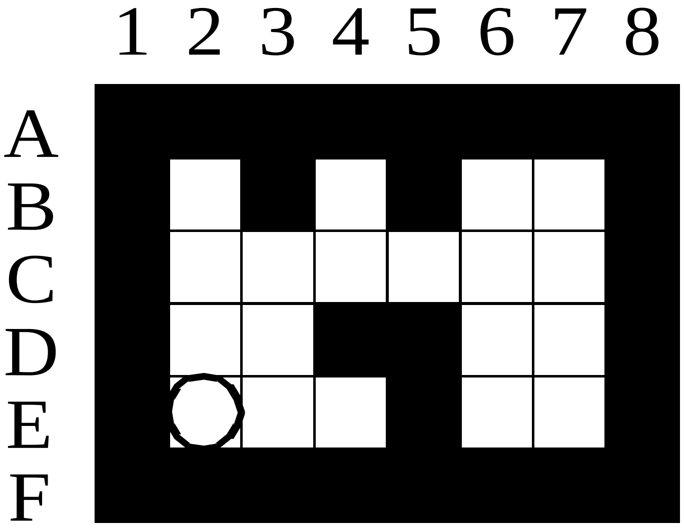
Mapping

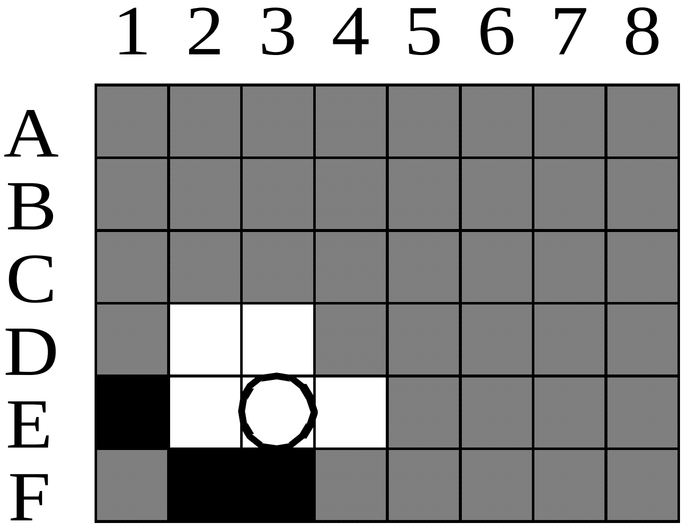




4-neighbor grid

Mapping

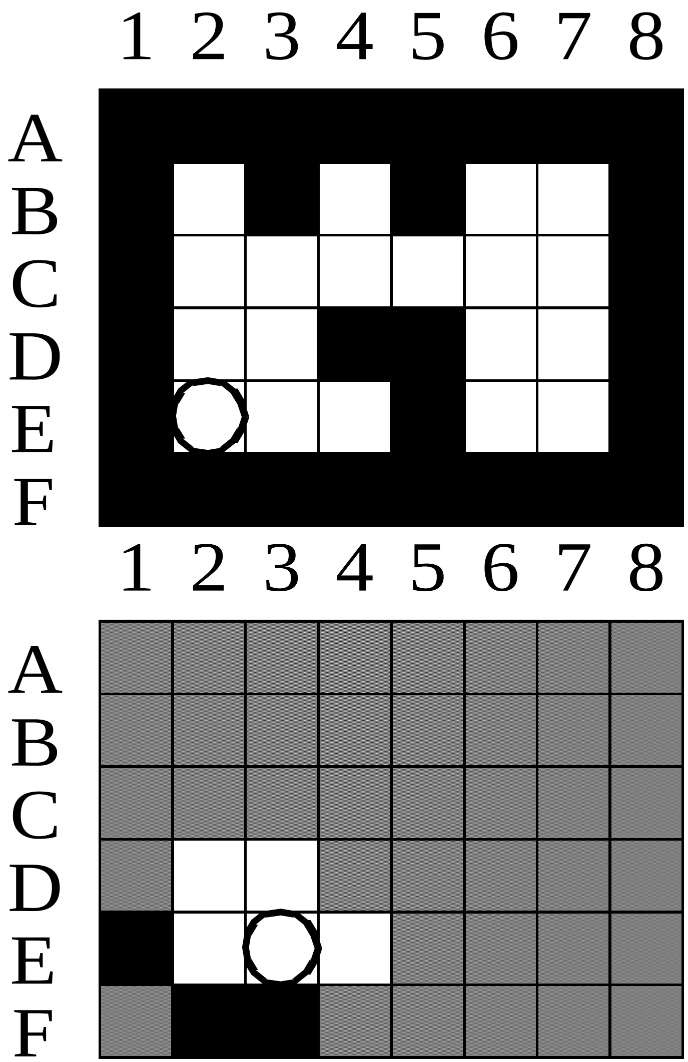




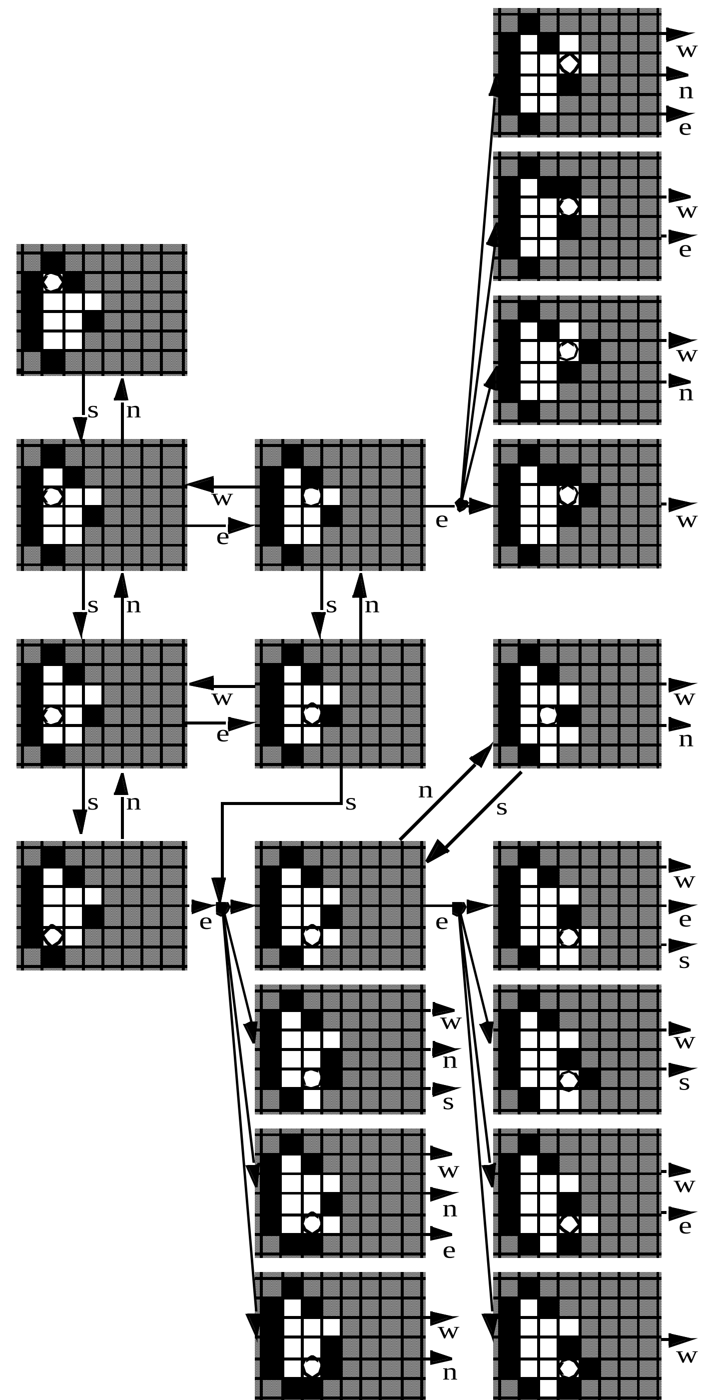
4-neighbor grid

Sven

Mapping

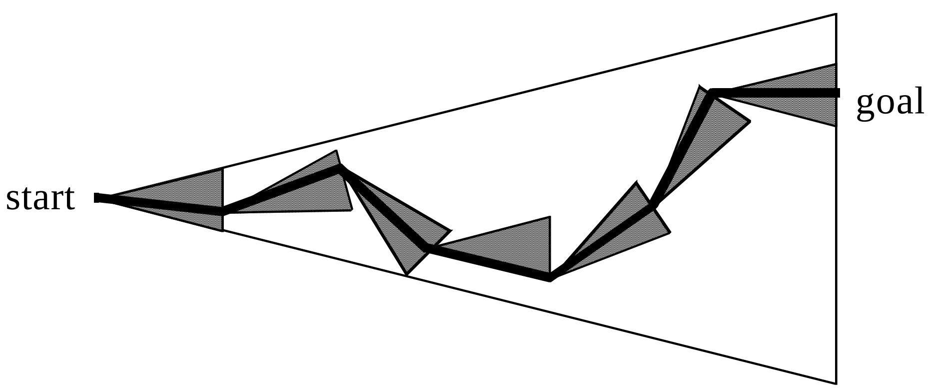


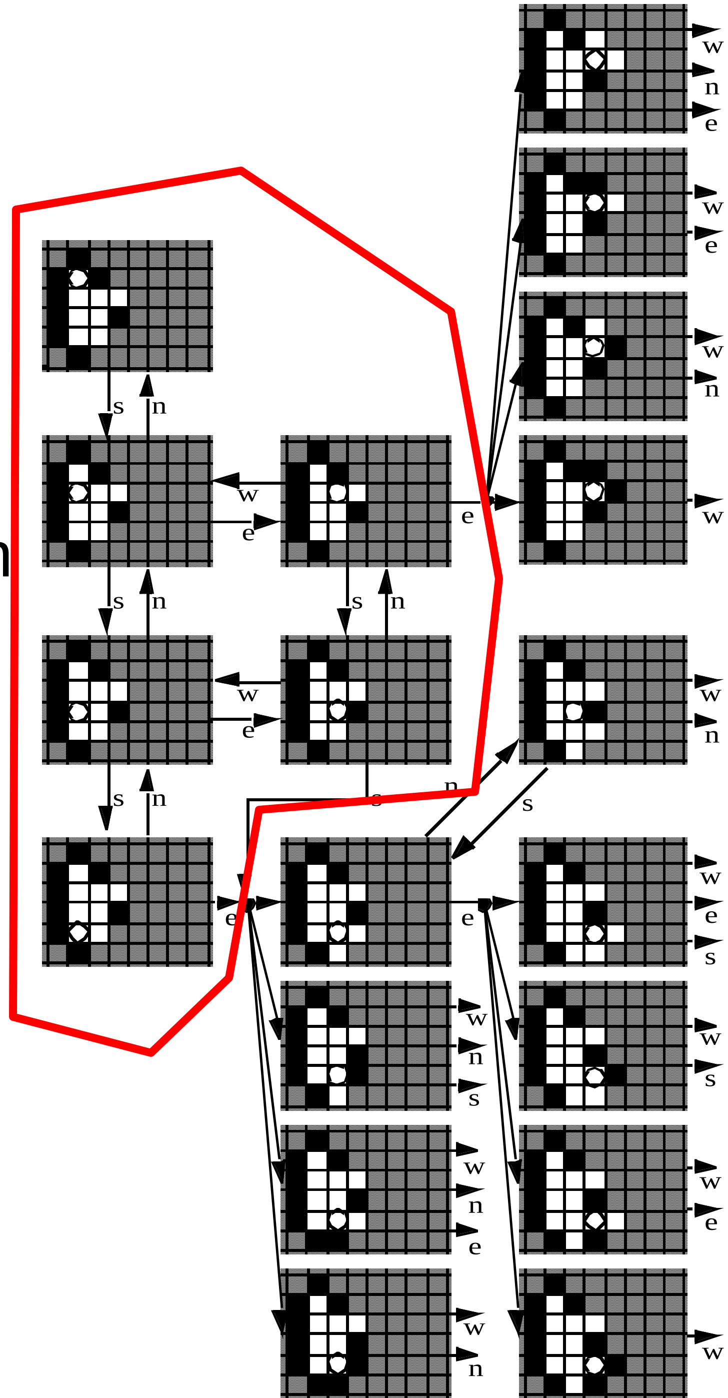
4-neighbor grid



Greedy Mapping

- Agent-centered search interleaves deterministic searches that result in a gain in information with action executions.

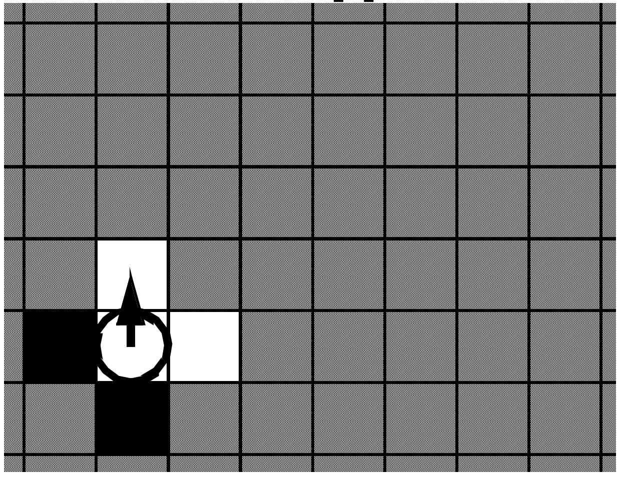




4-neighbor grid

Greedy Mapping

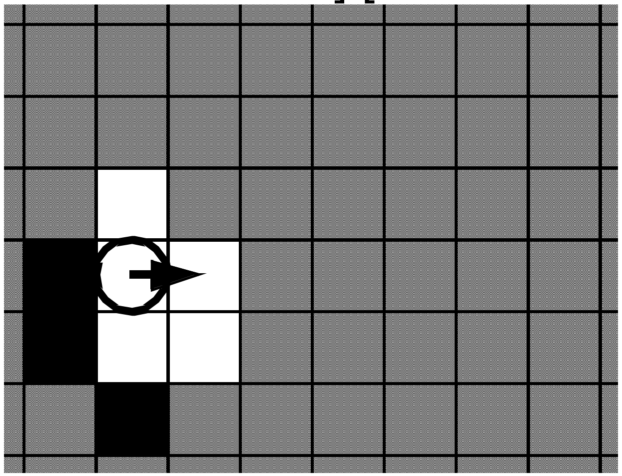
- Greedy mapping repeatedly makes the robot execute a shortest movement sequence to the closest informative unblocked cell, where an informative cell is one that allows the robot to observe the blockage status of at least one additional cell [Thrun et al., 1998] [Romero, Morales and Sucar, 2001].



4-neighbor grid

Greedy Mapping

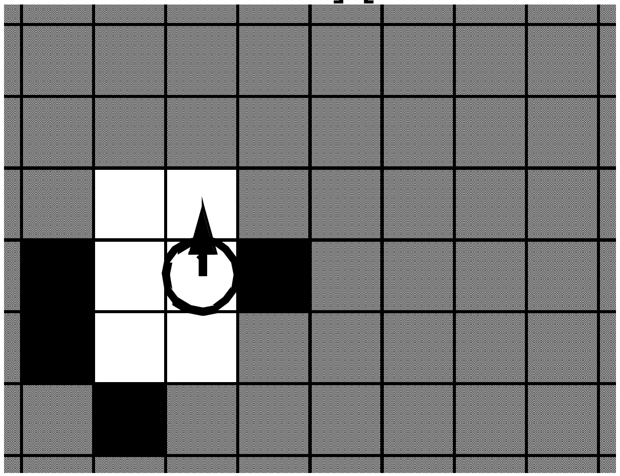
- Greedy mapping repeatedly makes the robot execute a shortest movement sequence to the closest informative unblocked cell, where an informative cell is one that allows the robot to observe the blockage status of at least one additional cell [Thrun et al., 1998] [Romero, Morales and Sucar, 2001].



4-neighbor grid

Greedy Mapping

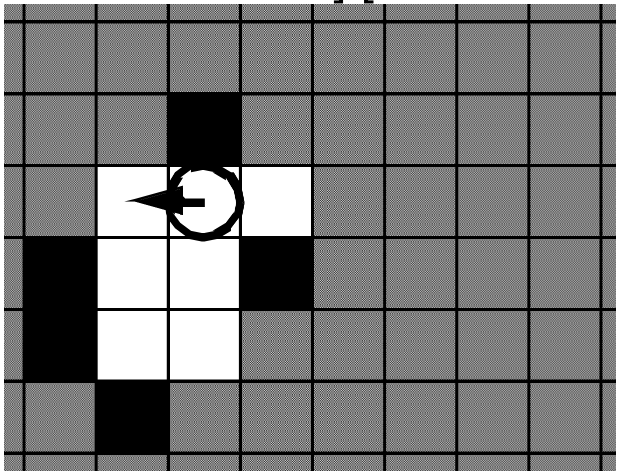
- Greedy mapping repeatedly makes the robot execute a shortest movement sequence to the closest informative unblocked cell, where an informative cell is one that allows the robot to observe the blockage status of at least one additional cell [Thrun et al., 1998] [Romero, Morales and Sucar, 2001].



4-neighbor grid

Greedy Mapping

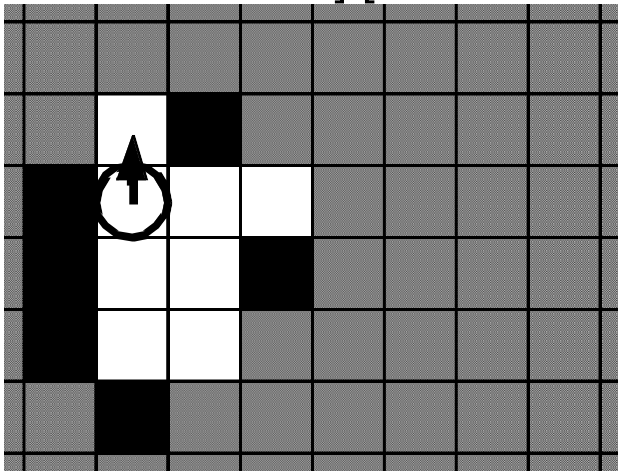
- Greedy mapping repeatedly makes the robot execute a shortest movement sequence to the closest informative unblocked cell, where an informative cell is one that allows the robot to observe the blockage status of at least one additional cell [Thrun et al., 1998] [Romero, Morales and Sucar, 2001].



4-neighbor grid

Greedy Mapping

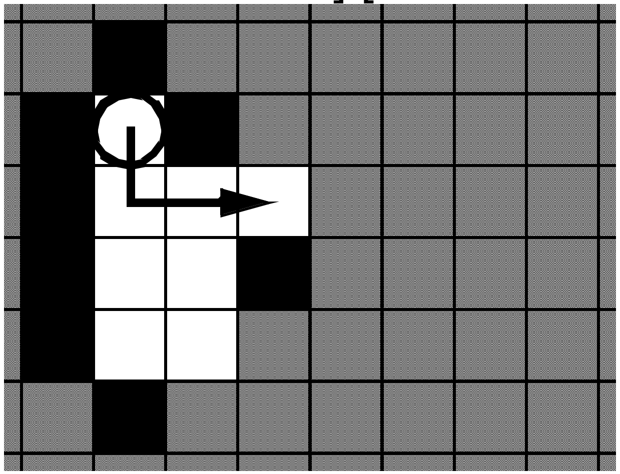
- Greedy mapping repeatedly makes the robot execute a shortest movement sequence to the closest informative unblocked cell, where an informative cell is one that allows the robot to observe the blockage status of at least one additional cell [Thrun et al., 1998] [Romero, Morales and Sucar, 2001].



4-neighbor grid

Greedy Mapping

- Greedy mapping repeatedly makes the robot execute a shortest movement sequence to the closest informative unblocked cell, where an informative cell is one that allows the robot to observe the blockage status of at least one additional cell [Thrun et al., 1998] [Romero, Morales and Sucar, 2001].



4-neighbor grid

Greedy Mapping

- Greedy mapping starts at some unblocked cell. It marks the robot cell (and perhaps other cells as well) as uninformative and then moves to the closest informative unblocked cell. It repeats the process until all unblocked cells are marked uninformative.

- Corollary [Tovey and Koenig, 2003]

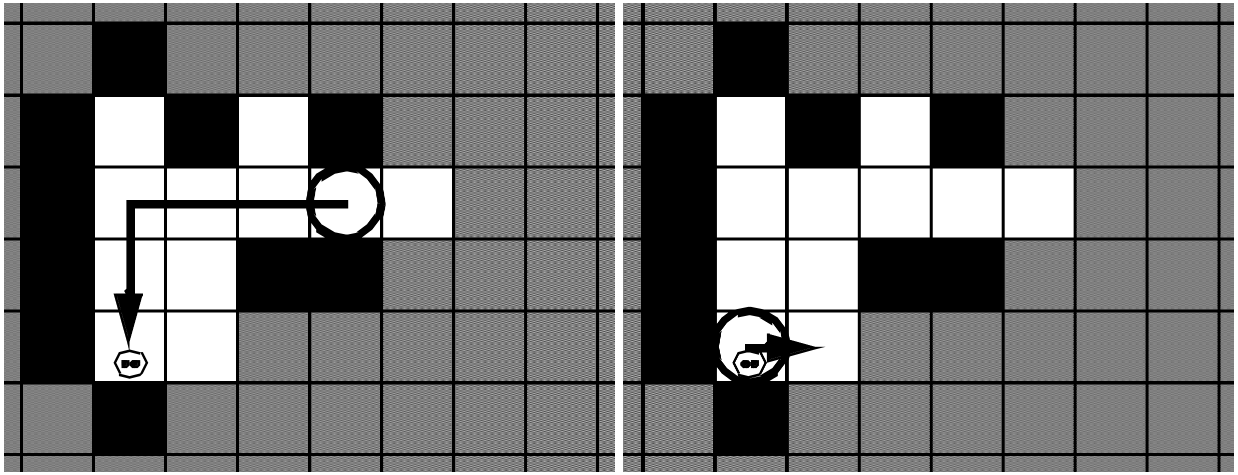
  The worst-case number of movements of greedy mapping is O(|V| log |V|), where |V| is the number of states (= unblocked cells).

Greedy Mapping

	Greedy mapping
Planning time	Low-order polynomial
Plan-execution time	Low-order polynomial

Greedy Mapping

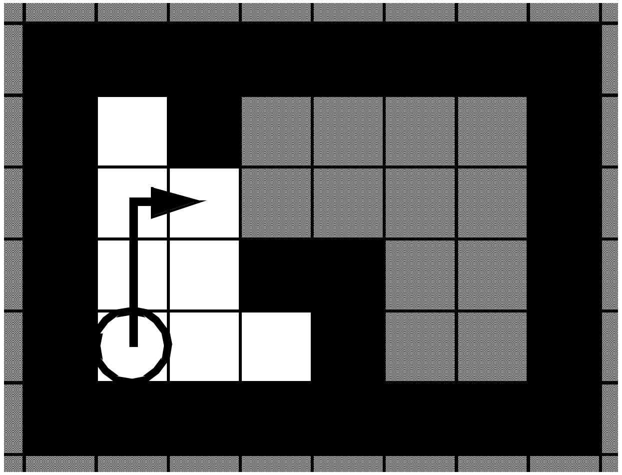
- Greedy mapping is reactive to changes in the robot cell. Thus, the robot does not need to move as instructed by greedy mapping.
- Other modules of a robot architecture can switch off greedy mapping and reactivate it later.



4-neighbor grid

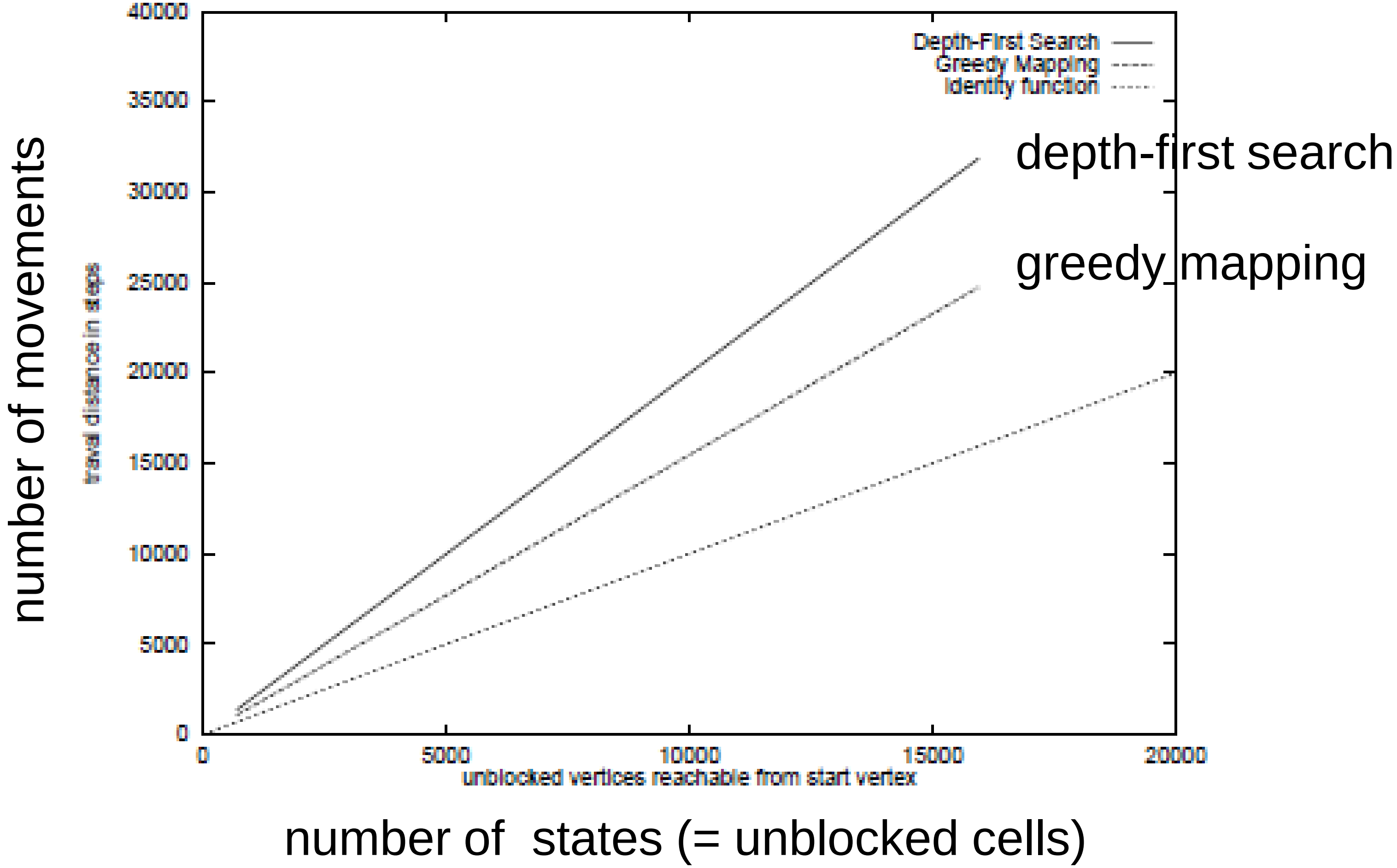
Greedy Mapping

- Greedy mapping is reactive to changes in the robot's knowledge of the terrain, independent of how the knowledge was obtained.
  - Greedy mapping immediately uses new terrain information, e.g. information provided by the user.

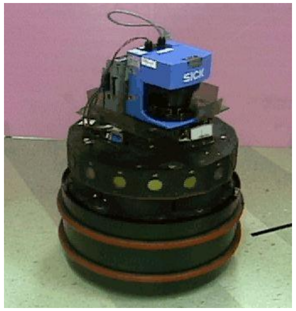


4-neighbor grid

Greedy Mapping



Greedy Mapping



20 feet

28 feet

Planning Problems and Strategies

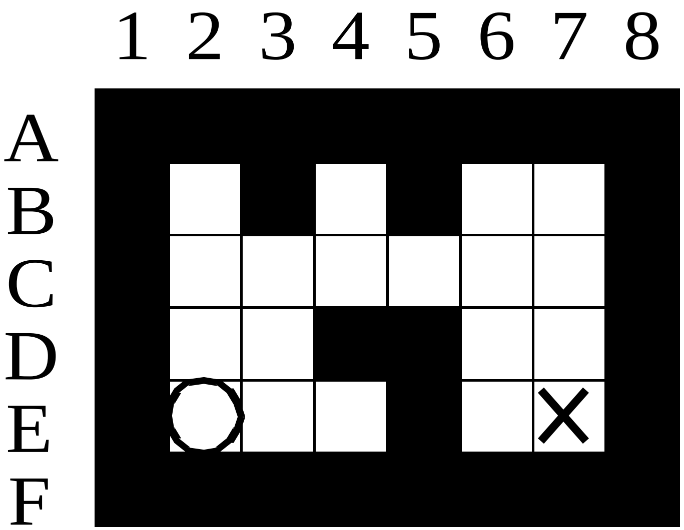
- Greedy Agent-Centered Search
- Three Robot-Navigation Problems and Approaches
  - Localization using Agent-Centered Search:
    Greedy Localization
  - Mapping using Agent-Centered Search:
    Greedy Mapping
  - Stationary Target Search in Unknown Terrain
    using Assumption-Based Planning:
    Planning with the Freespace Assumption
- Summary
  - Agent-Centered Search
  - Planning with the Freespace Assumption
  - Real-Time Search

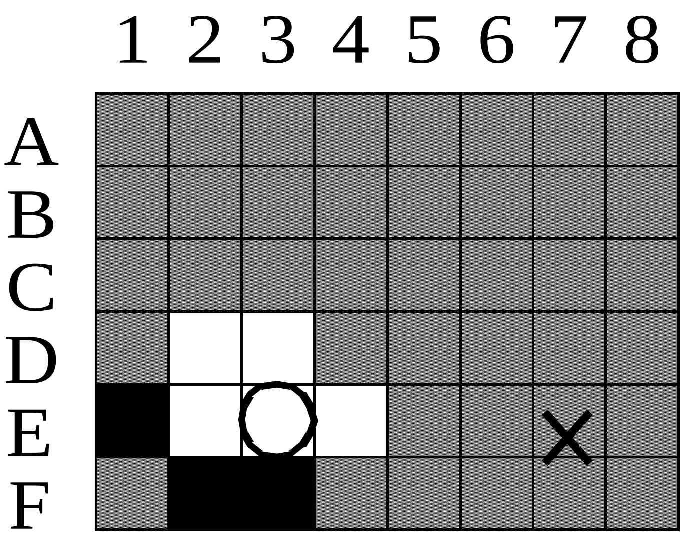
Stationary Target

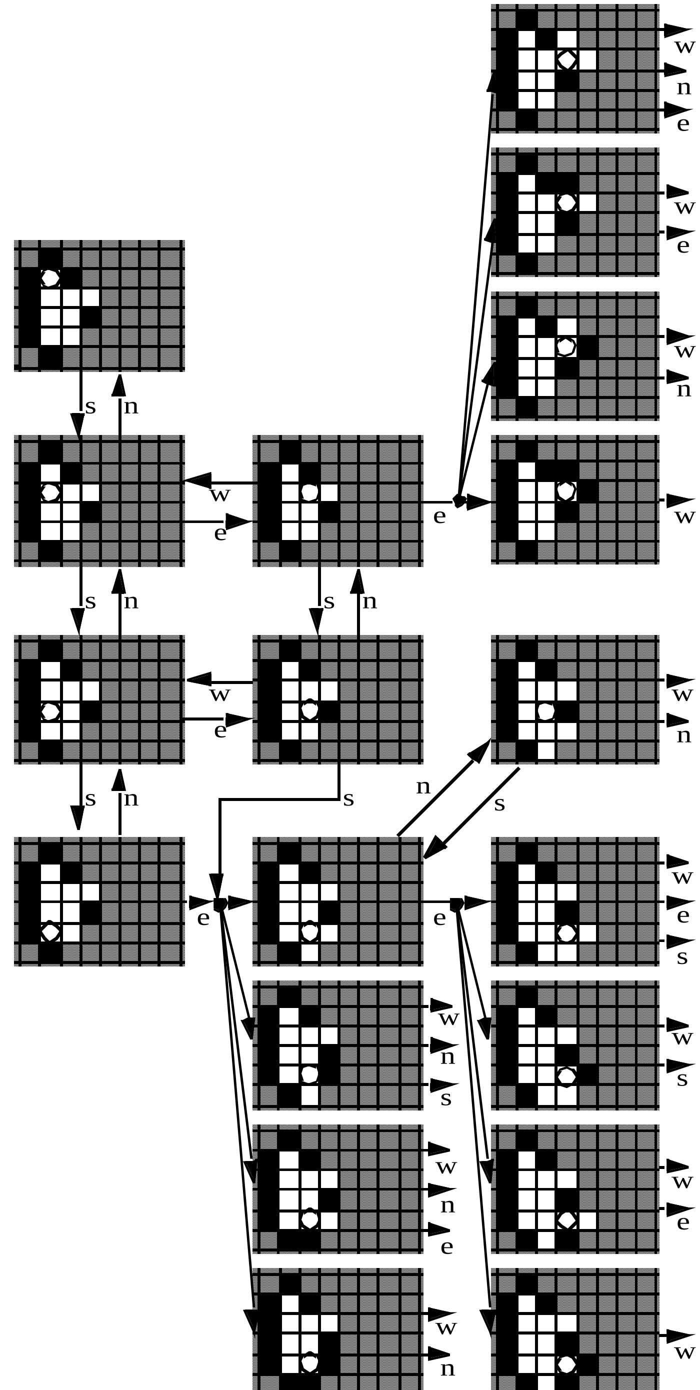
- Stationary target-search navigates to a stationary target cell with no a priori given map, always knowing the robot cell. (Stationary target search is often called goal-directed navigation.)

Sven

Stationary Target



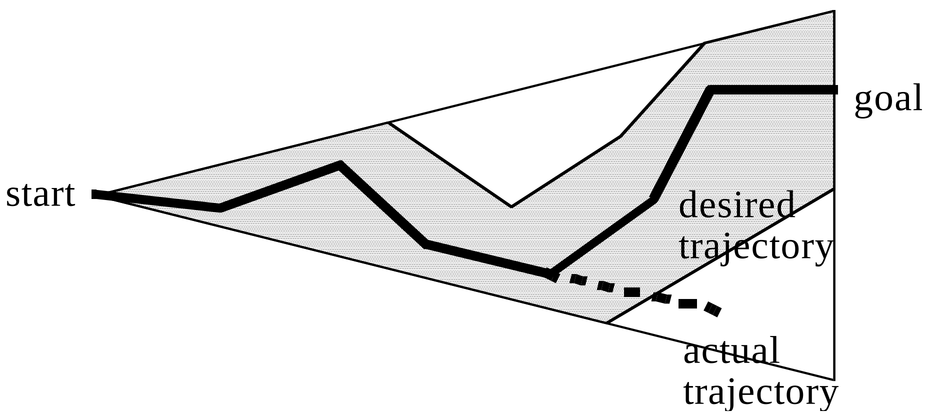


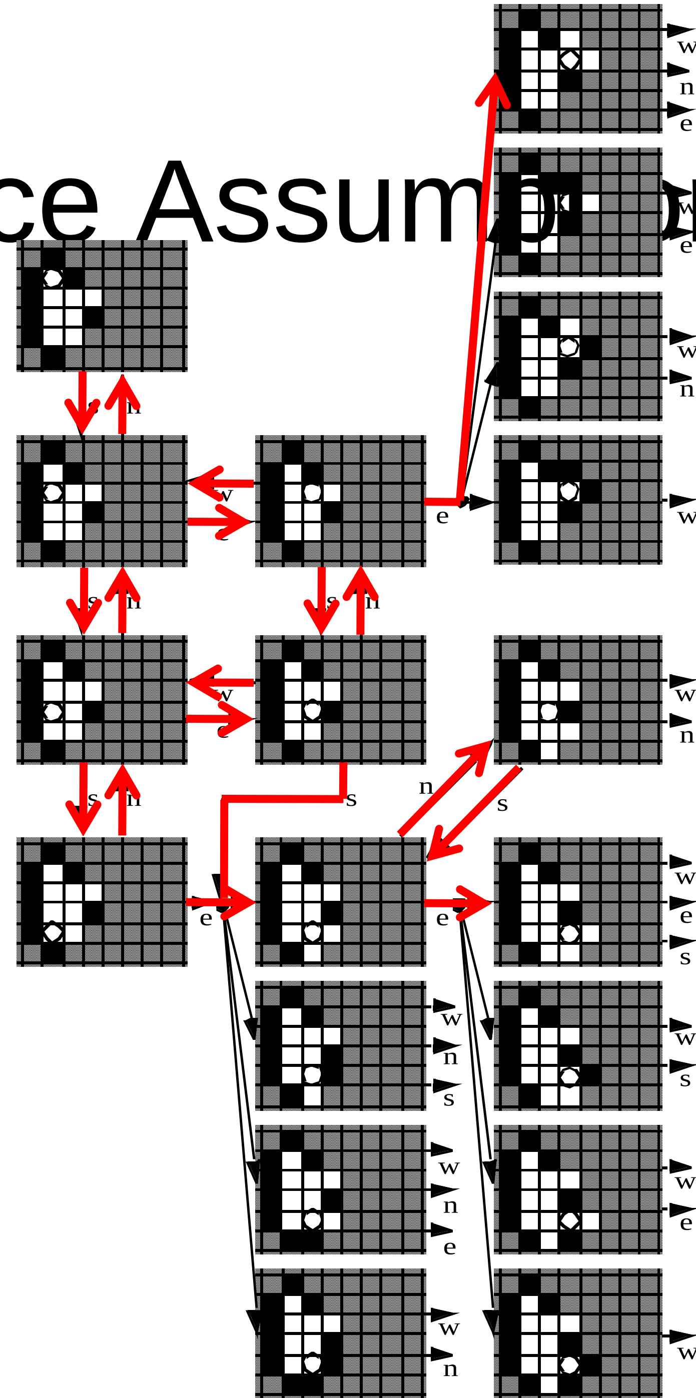


4-neighbor grid

Pl w the Freespace Assum on

- Assumption-based planning interleaves deterministic searches resulting from making assumptions about action outcomes with action executions.

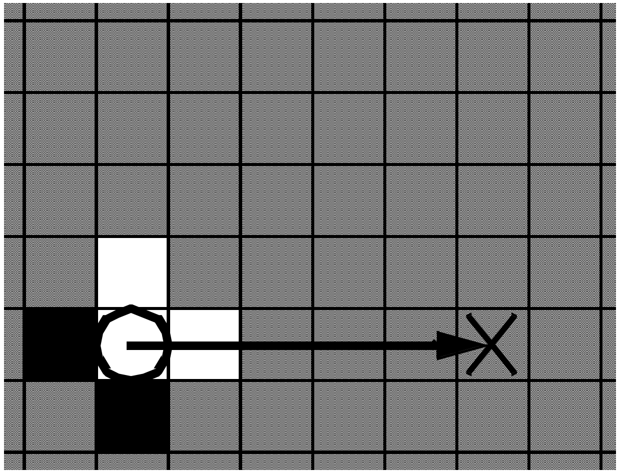




4-neighbor grid

Pl w the Freespace Assumption

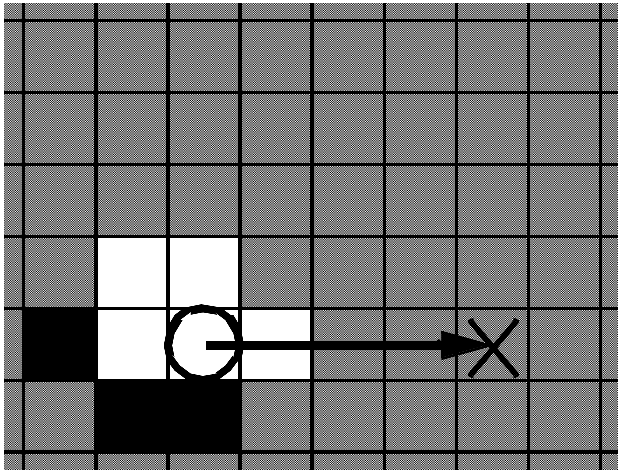
- Planning with the freespace assumption repeatedly makes the robot execute a shortest movement sequence to the goal under the assumption that cells with unknown blockage status are unblocked [Brumitt and Stentz, 1998] [Hebert, McLachlan, Chang, 1999] [Stentz and Hebert, 1995].



4-neighbor grid

Pl w the Freespace Assumption

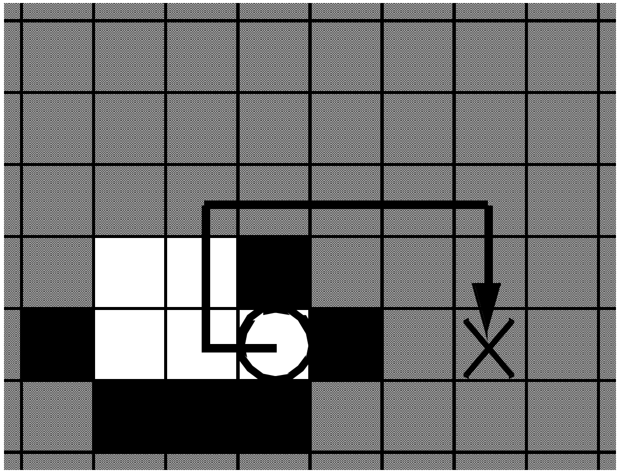
- Planning with the freespace assumption repeatedly makes the robot execute a shortest movement sequence to the goal under the assumption that cells with unknown blockage status are unblocked [Brumitt and Stentz, 1998] [Hebert, McLachlan, Chang, 1999] [Stentz and Hebert, 1995].



4-neighbor grid

Pl w the Freespace Assumption

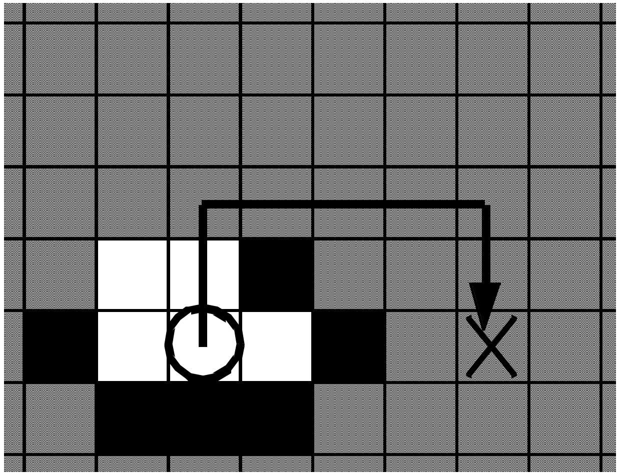
- Planning with the freespace assumption repeatedly makes the robot execute a shortest movement sequence to the goal under the assumption that cells with unknown blockage status are unblocked [Brumitt and Stentz, 1998] [Hebert, McLachlan, Chang, 1999] [Stentz and Hebert, 1995].



4-neighbor grid

Pl w the Freespace Assumption

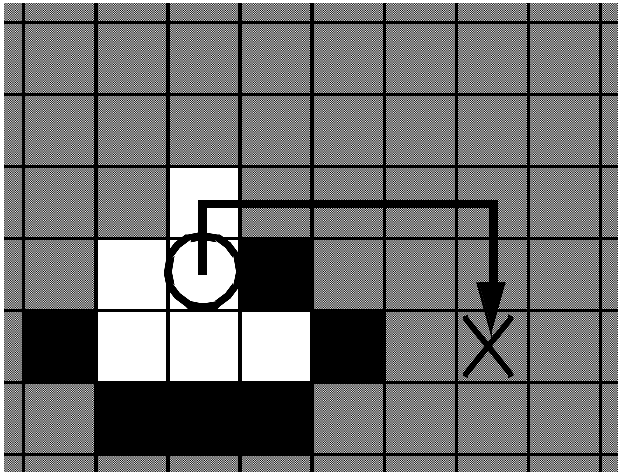
- Planning with the freespace assumption repeatedly makes the robot execute a shortest movement sequence to the goal under the assumption that cells with unknown blockage status are unblocked [Brumitt and Stentz, 1998] [Hebert, McLachlan, Chang, 1999] [Stentz and Hebert, 1995].



4-neighbor grid

Pl w the Freespace Assumption

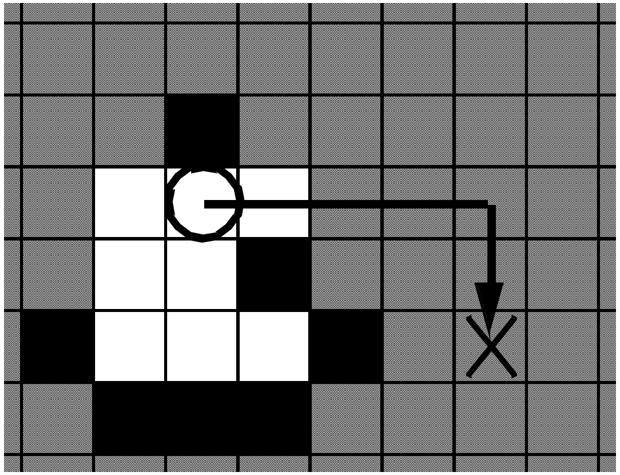
- Planning with the freespace assumption repeatedly makes the robot execute a shortest movement sequence to the goal under the assumption that cells with unknown blockage status are unblocked [Brumitt and Stentz, 1998] [Hebert, McLachlan, Chang, 1999] [Stentz and Hebert, 1995].



4-neighbor grid

Pl w the Freespace Assumption

- Planning with the freespace assumption repeatedly makes the robot execute a shortest movement sequence to the goal under the assumption that cells with unknown blockage status are unblocked [Brumitt and Stentz, 1998] [Hebert, McLachlan, Chang, 1999] [Stentz and Hebert, 1995].



4-neighbor grid

Pl w the Freespace Assumption

- Theorem [Mudgal, Tovey and Koenig, 2004]

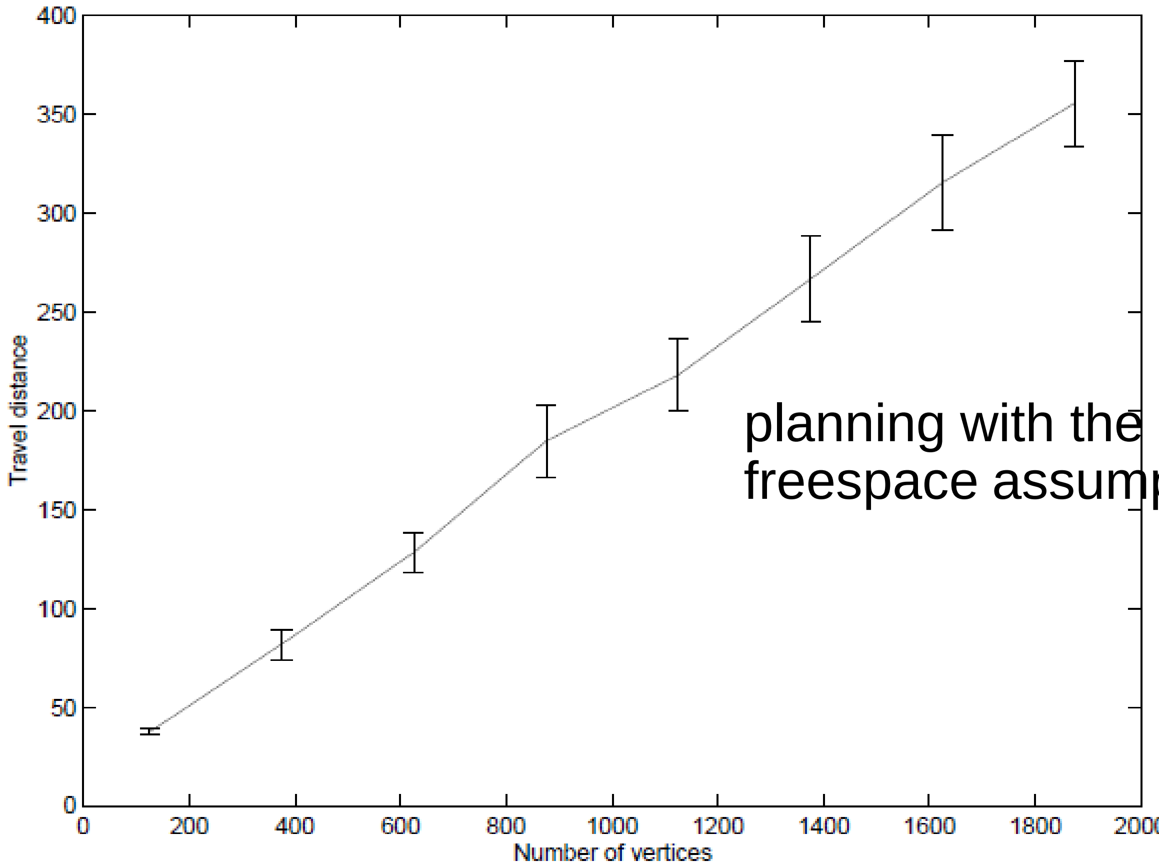
  The worst-case number of movements of planning with the freespace assumption is O(|V| log |V|), where |V| is the number of states (= unblocked cells).

Pl w the Freespace Assumption

	Planning with the freespace assumption
Planning time	Low-order polynomial
Plan-execution time	Low-order polynomial

Pl w the Freespace Assumption

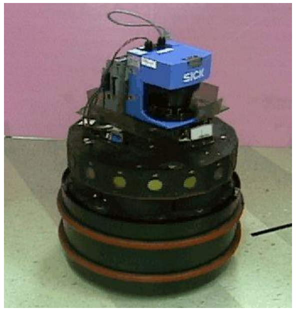
number of movements



planning with the
freespace assumption

number of states (= unblocked cells)

Pl w the Freespace Assumption



20 feet

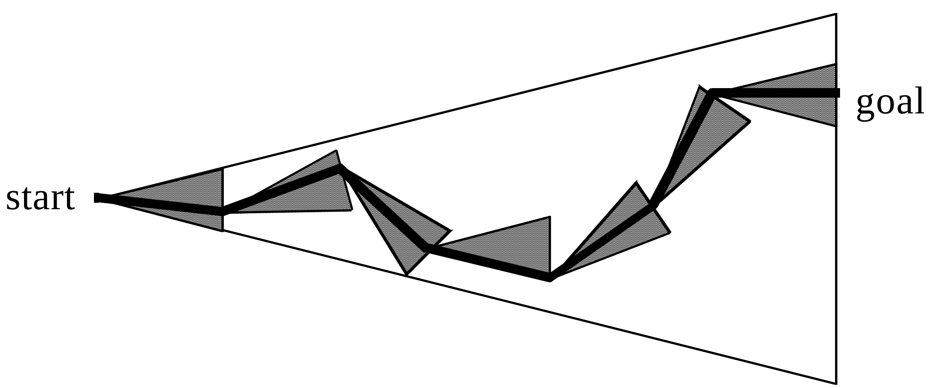
28 feet

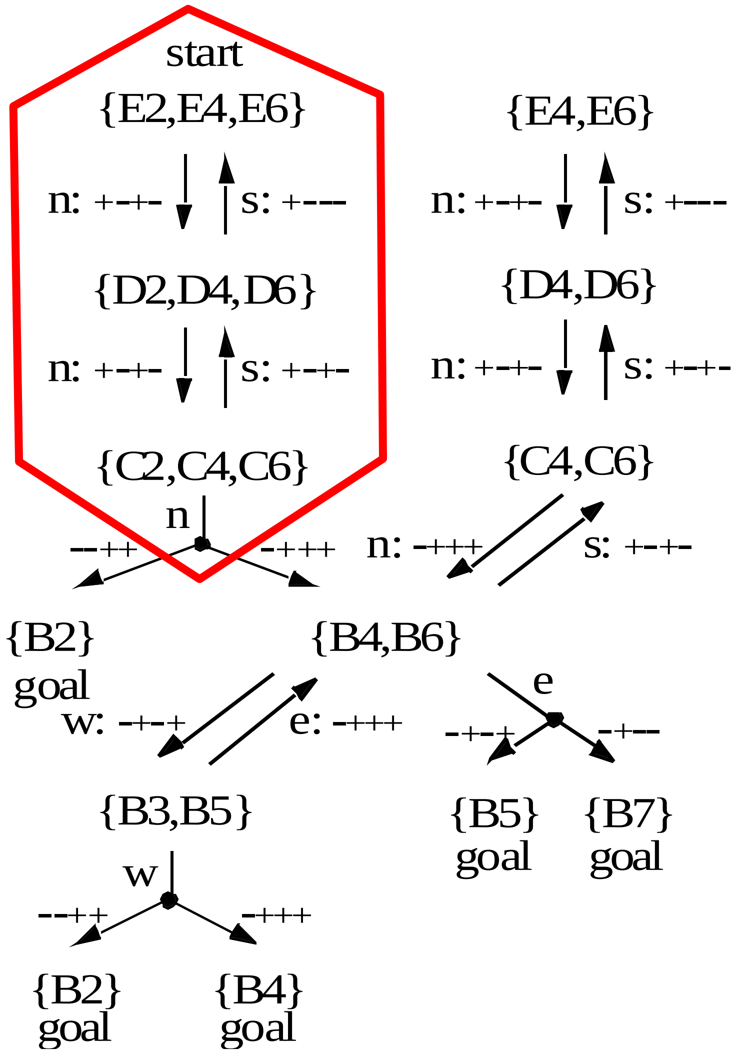
Planning Problems and Strategies

- Greedy Agent-Centered Search
- Three Robot-Navigation Problems and Approaches
  - Localization using Agent-Centered Search:
    Greedy Localization
  - Mapping using Agent-Centered Search:
    Greedy Mapping
  - Stationary Target Search in Unknown Terrain
    using Assumption-Based Planning:
    Planning with the Freespace Assumption
- Summary
  - Agent-Centered Search
  - Planning with the Freespace Assumption
  - Real-Time Search

Agent-Centered Search

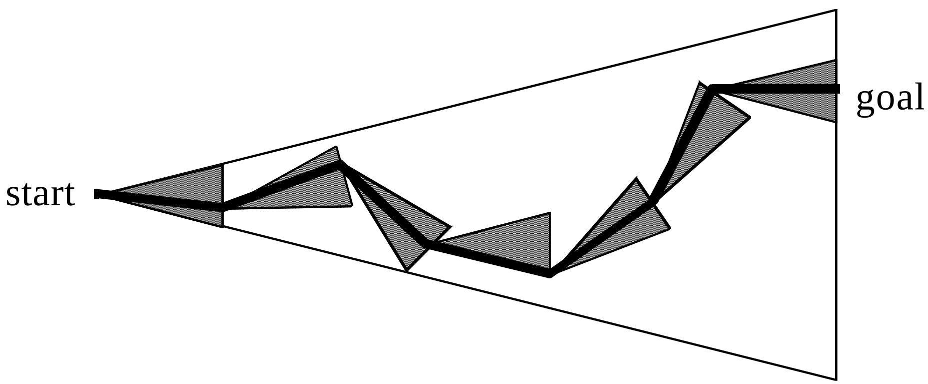
- Agent-centered search interleaves deterministic searches that result in a gain in information with action executions.

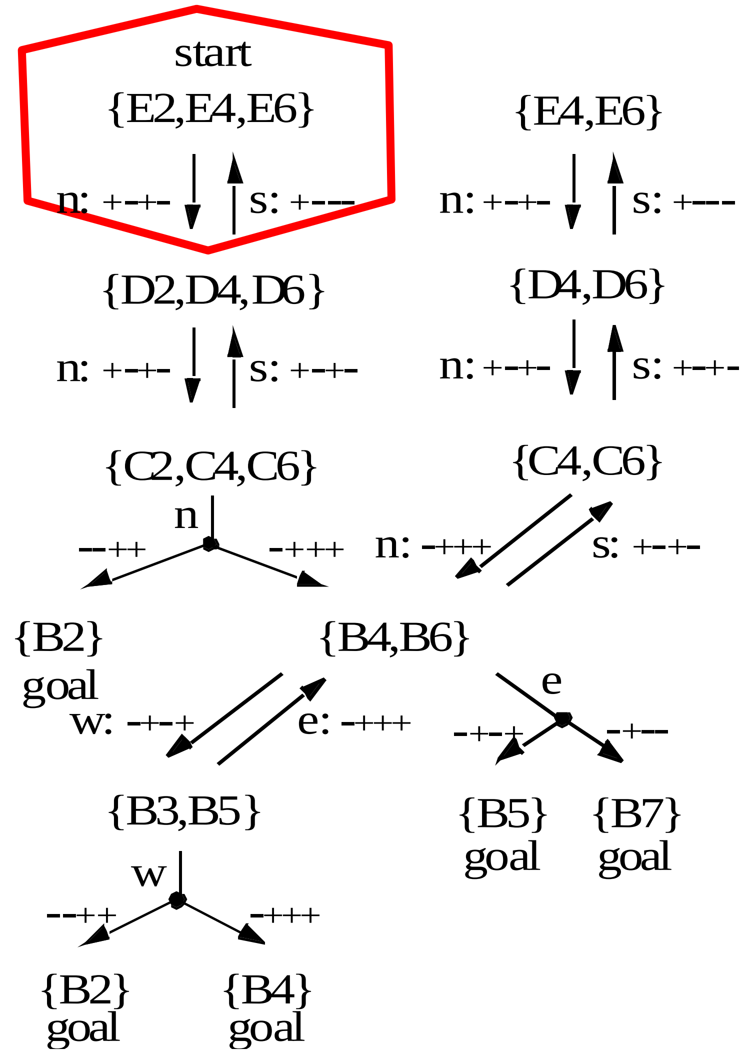




Real-Time Search

- Real-time search interleaves deterministic searches ~~that result in a gain in information~~ with action executions.

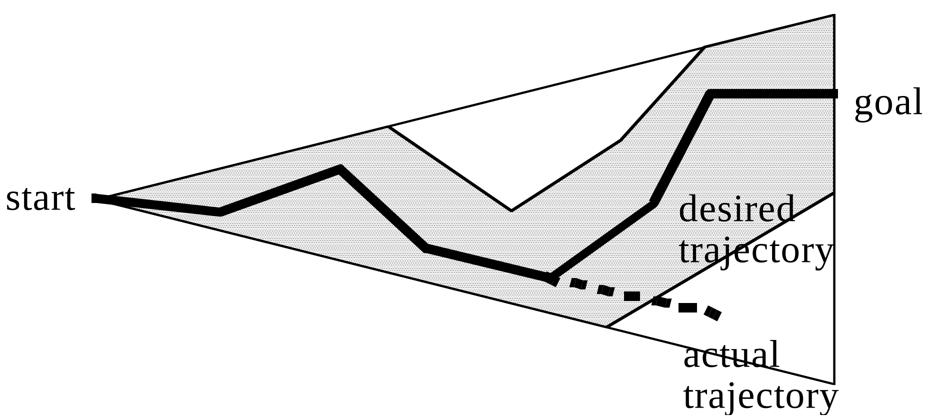


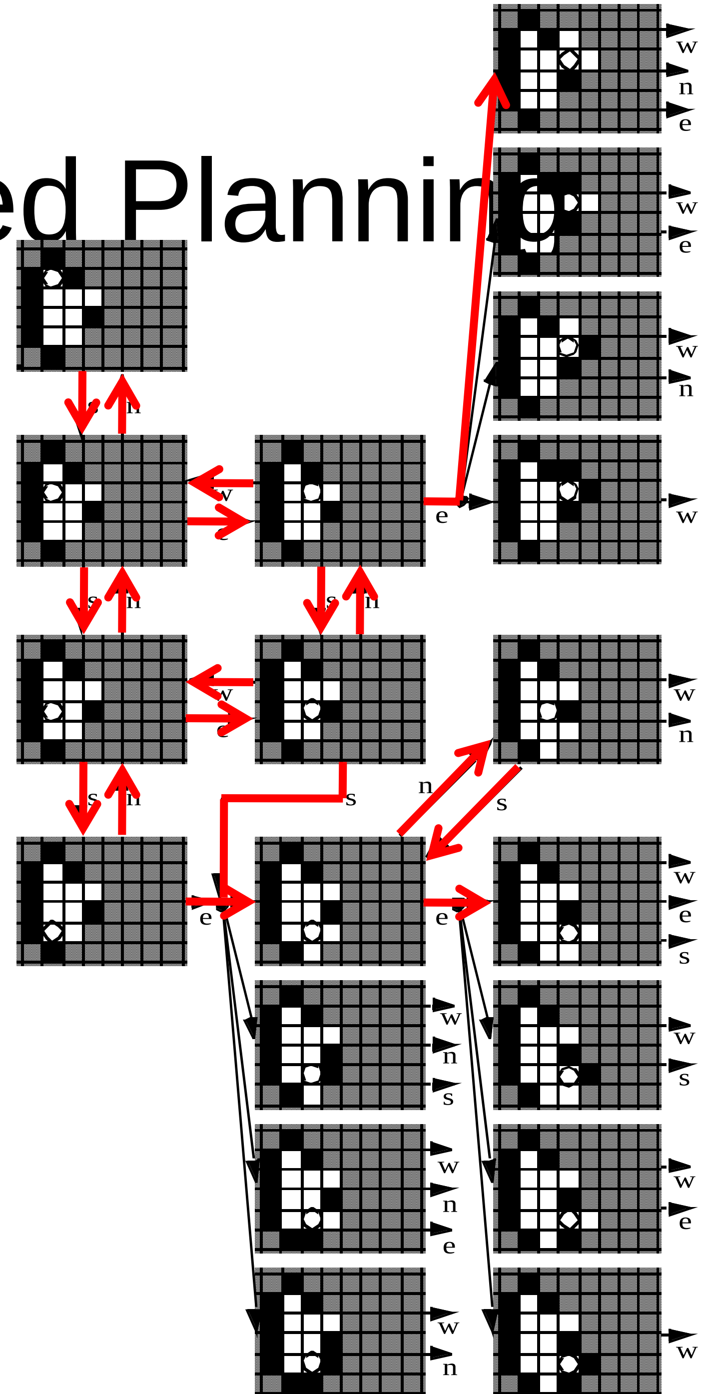


Sven

Assumption-Based Planning

- Assumption-based planning interleaves deterministic searches resulting from making assumptions about action outcomes with action executions.



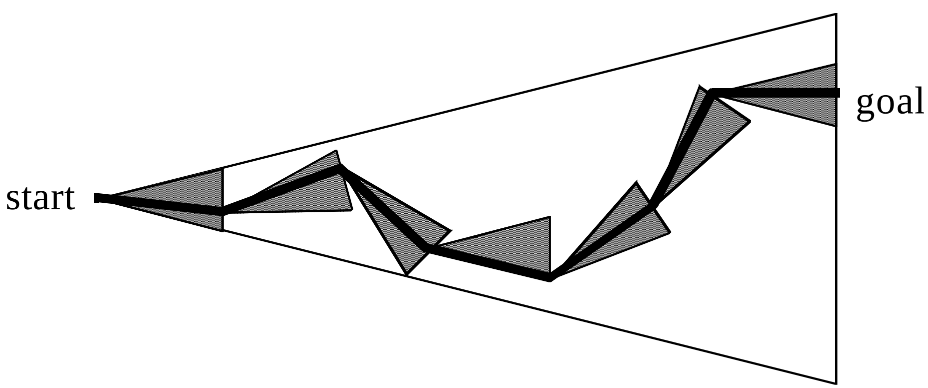


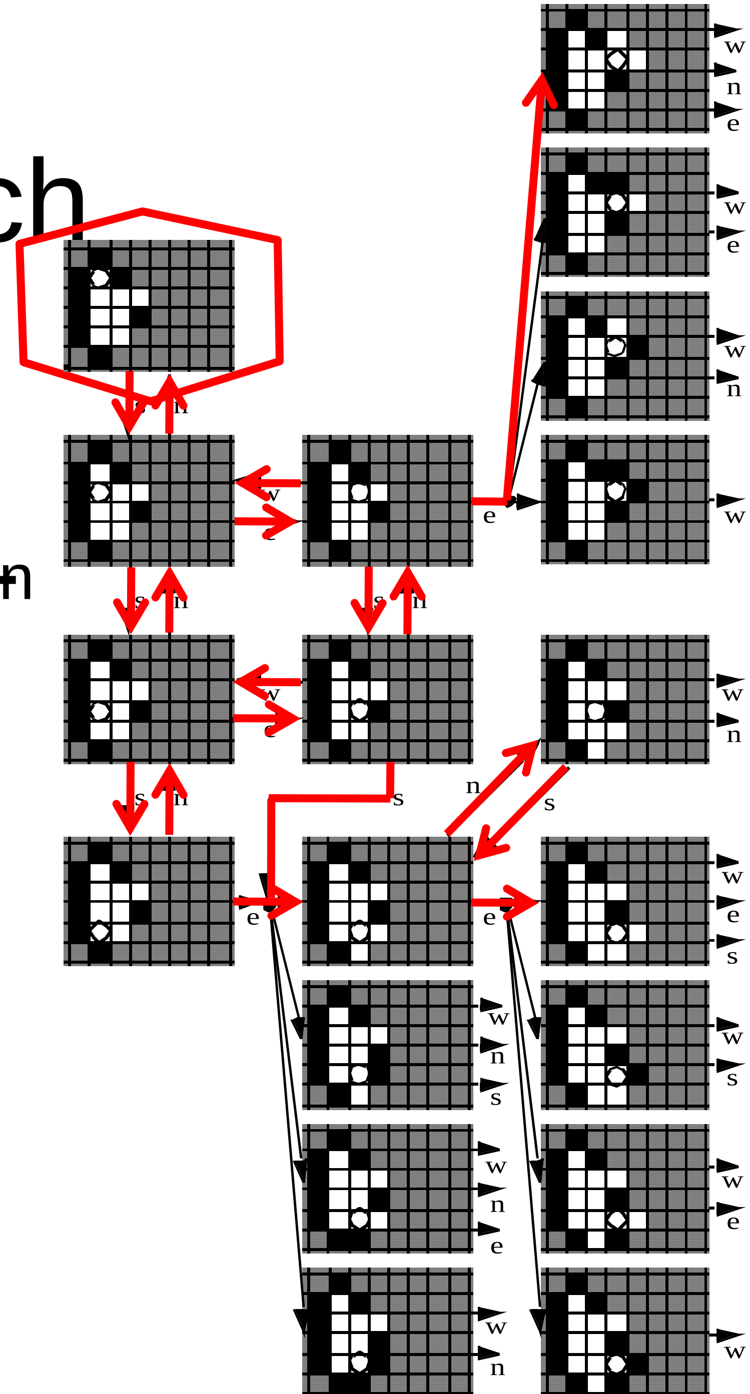
4-neighbor grid

Sven

Real-Time Search

- Real-time search interleaves deterministic searches ~~that result in a gain in information~~ with action executions.





4-neighbor grid

Issues

- Agent-centered search
  - How to find similar plans efficiently?
  - How much to plan…
    - to guarantee that the objective is achieved and
    - to trade off well between planning and plan-execution time?
- Assumption-based planning
  - How to find similar plans efficiently?
  - Which assumptions to make
    - to guarantee that the objective is achieved and
    - to trade off well between planning and plan-execution time?

Table of Contents

- Modeling Planning Domains
  - Graphs, MDPs
- Planning Problems and Strategies
  - Localization, Mapping, Navigation in Unknown Terrain
  - Agent-Centered Search, Assumptive Planning
- Efficient Implementations of Planning Strategies
  - Incremental Heuristic Search

15 Minute Break

  - Real-Time Heuristic Search
  - Planning with Preferences on Uncertainty
  - Planning with Varying Abstractions

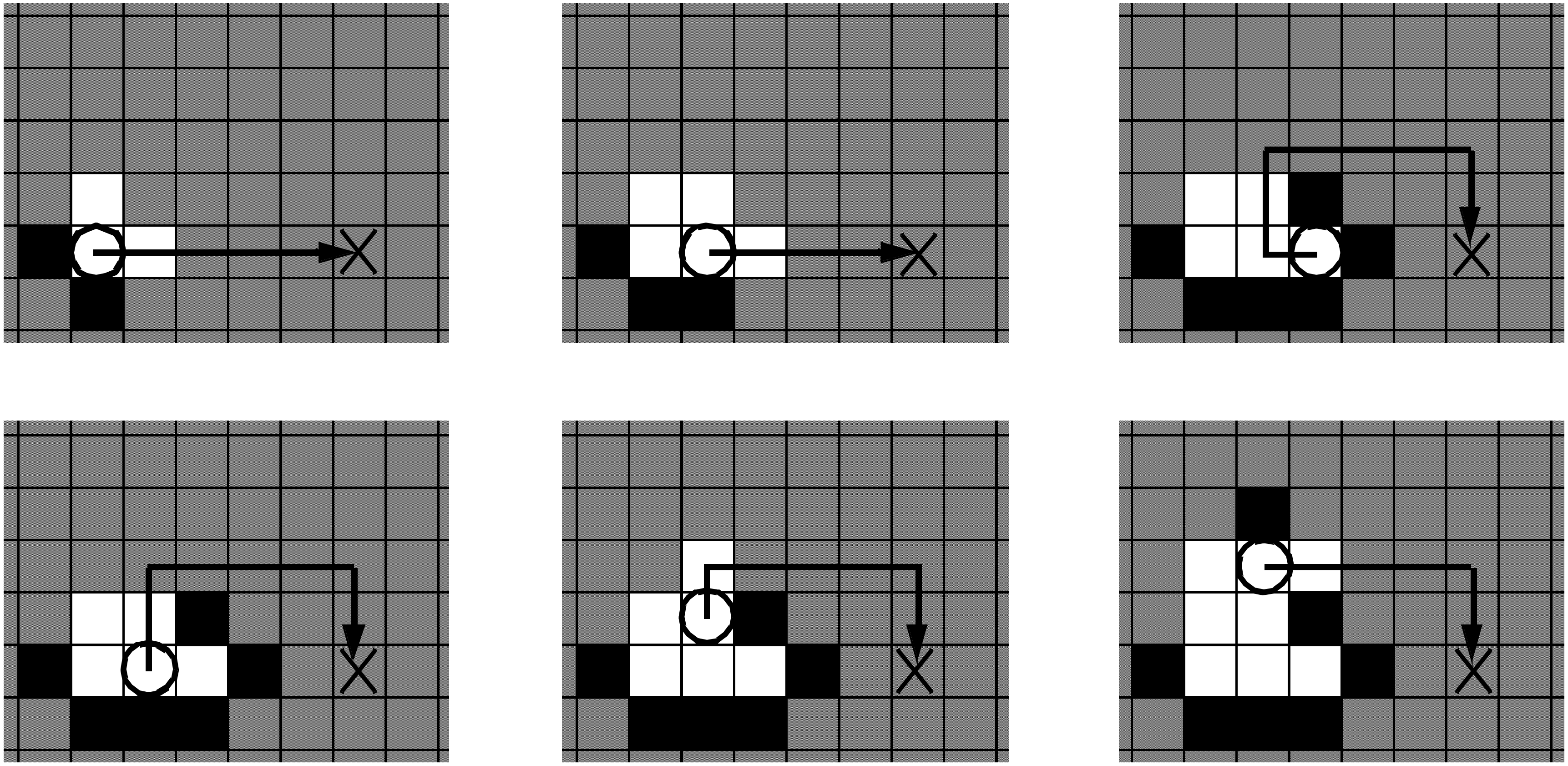
Incremental Heuristic Search

search task 1	slightly different search task 2	slightly different search task 3

Stationary Target

- Stationary target search navigates to a stationary target cell with no a priori given map, always knowing the robot cell.

Stationary Target



4-neighbor grid

Incremental Heuristic Search

- Incremental heuristic search speeds up A* searches for a sequence of similar search problems by exploiting experience with earlier search problems in the sequence. It finds shortest paths.
- In the worst case, incremental heuristic search cannot be more efficient than A* searches from scratch [Nebel and Koehler 1995].

Incremental Heuristic Search

search task 1	slightly different search task 2	slightly different search task 3

search task 1	slightly different search task 2	slightly different search task 3	slightly different search task 4

Incremental Heuristic Search

- Fringe Saving A* (FSA*)
- Adaptive A* (AA*)
- Lifelong Planning A* (LPA*), D* Lite and Minimax LPA*
- Comparison of D* Lite and Adaptive A*
- Eager and Lazy Moving-Target Adaptive A* (MTAA*)
- Anytime Replanning A* (ARA*)
- Anytime D*

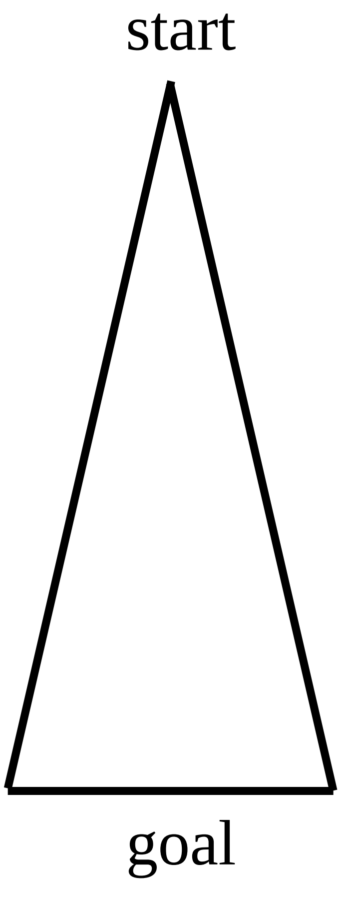
planning time per expansion increases

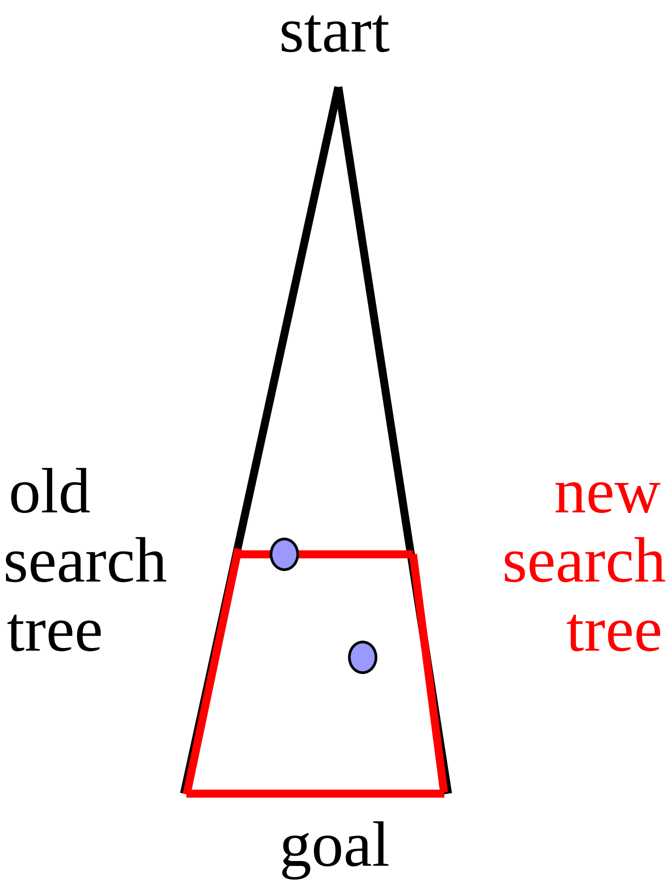
number of expansions decreases

Fringe Saving A* (FSA*)

- Fringe Saving A* (FSA*) [Sun and Koenig, 2007] speeds up A* searches for a sequence of similar search problems by starting each A* search at the point where it could differ from the previous one.
- FSA* is similar to but faster than iA* [Yap, unpublished].

Fringe Saving A* (FSA*)





Fringe Saving A* (FSA*)

- Seventh and last iteration of A*

order of expansions

			4	5	6
		3	1		(7)
			2		

g-values

			2	3	4
		2	1	2	3
	2	1	0		4
		2	1		

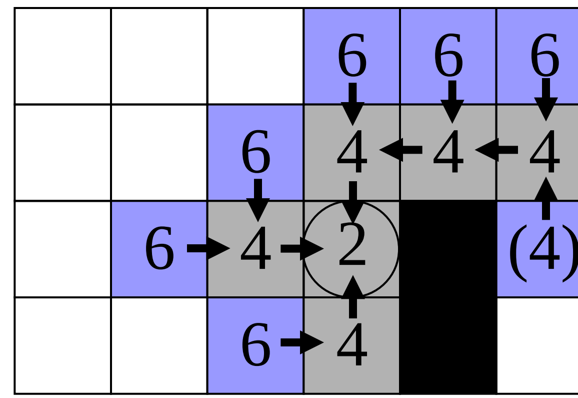
+

h-values

7	6	5	4	3	2
6	5	4	3	2	1
5	4	3	2	1	0
6	5	4	3	2	1

=

f-values



cost of the shortest path in the search tree from the start to the given state

generated but not expanded state (OPEN list)

expanded state (CLOSED list)

4-neighbor grid

order of expansions

			4		
		3	1		
			2		

Fringe Saving A* (FSA*)

- One cell becomes blocked

			2	3	4
		2	1		3
	2	1	0		4
		2	1		

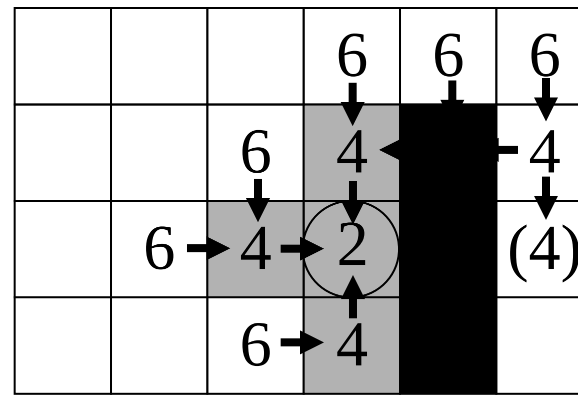
g-values

+

7	6	5	4	3	2
6	5	4	3	2	1
5	4	3	2	1	0
6	5	4	3	2	1

h-values

=



f-values

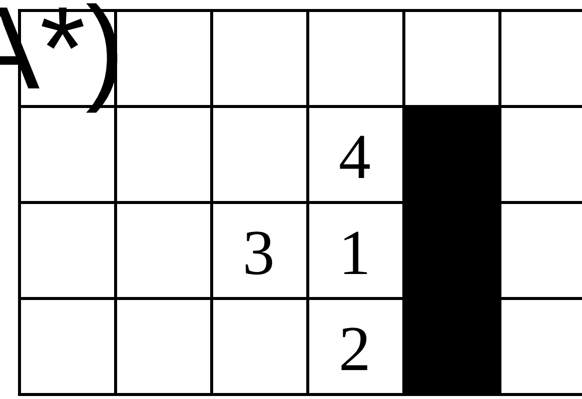
cost of the shortest path
In the search tree from the
start to the given state

generated but not expanded state (OPEN list)
expanded state (CLOSED list)

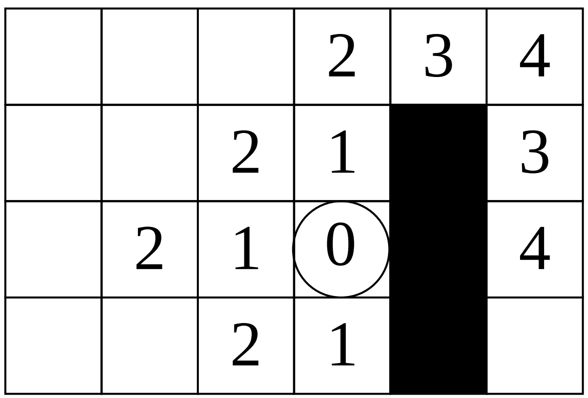
4-neighbor grid

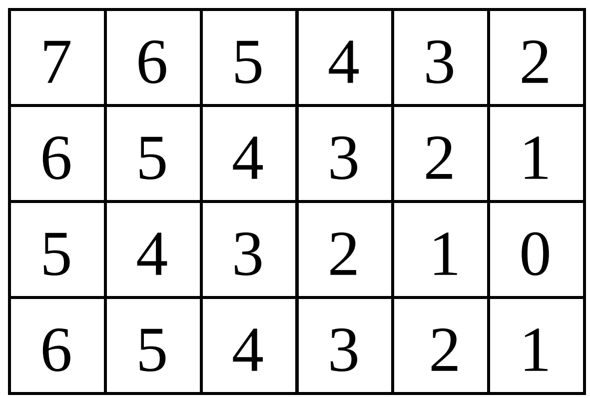
Fringe Saving A* (FSA*)

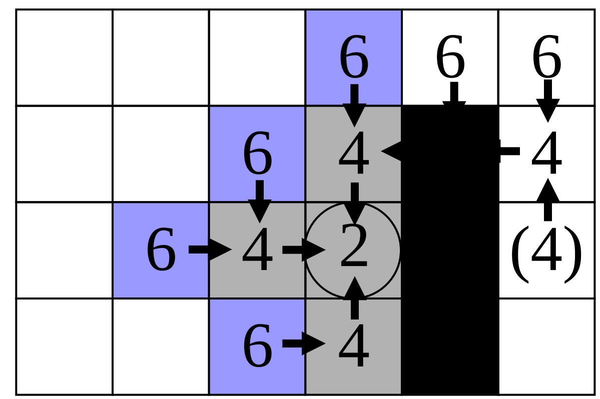
order of expansions



- One cell becomes blocked







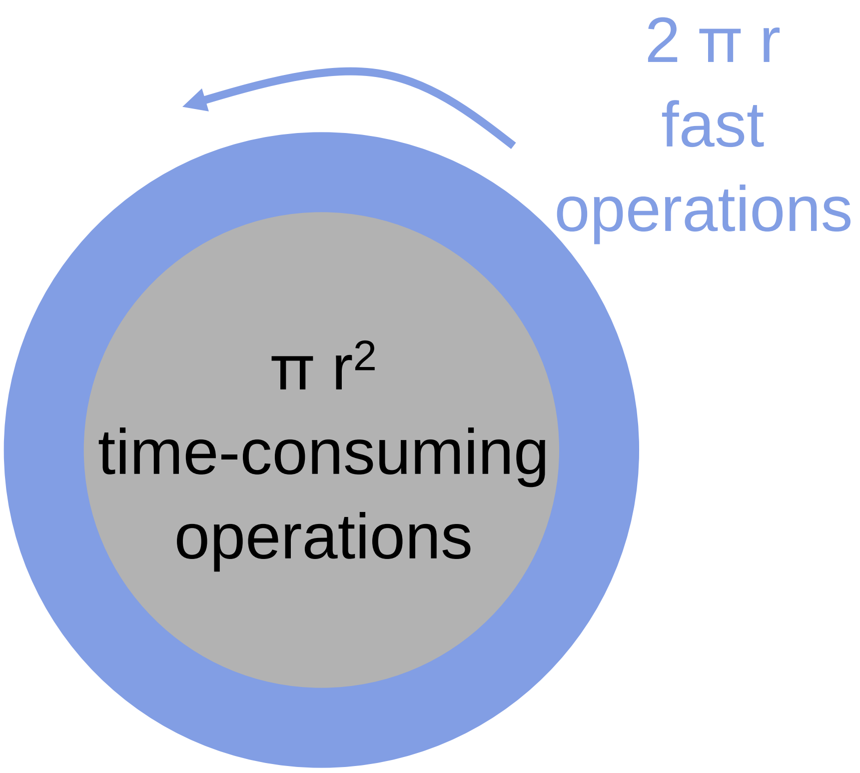
g-values + h-values = f-values

cost of the shortest path
In the search tree from the
start to the given state

- generated but not expanded state (OPEN list)
- expanded state (CLOSED list)

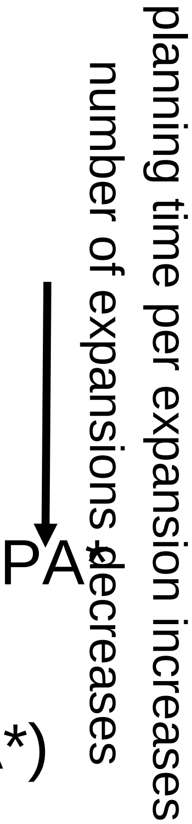
4-neighbor grid

Fringe Saving A* (FSA*)



Incremental Heuristic Search

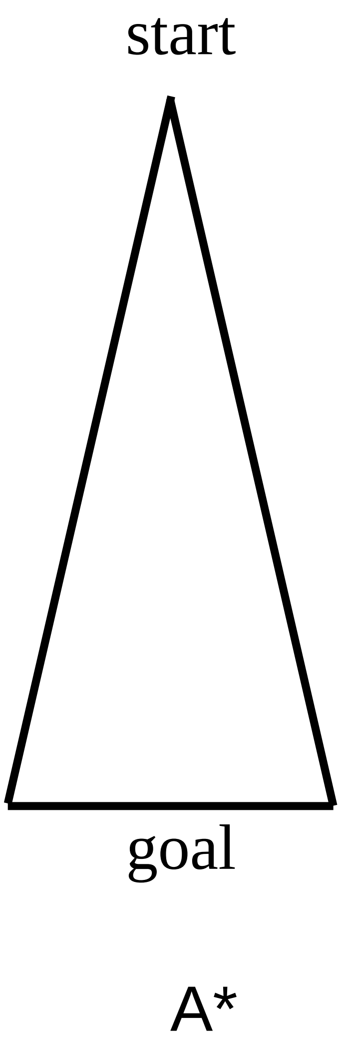
- Fringe Saving A* (FSA*)
- Adaptive A* (AA*)
- Lifelong Planning A* (LPA*), D* Lite and Minimax LPA*
- Comparison of D* Lite and Adaptive A*
- Eager and Lazy Moving-Target Adaptive A* (MTAA*)
- Anytime Replanning A* (ARA*)
- Anytime D*

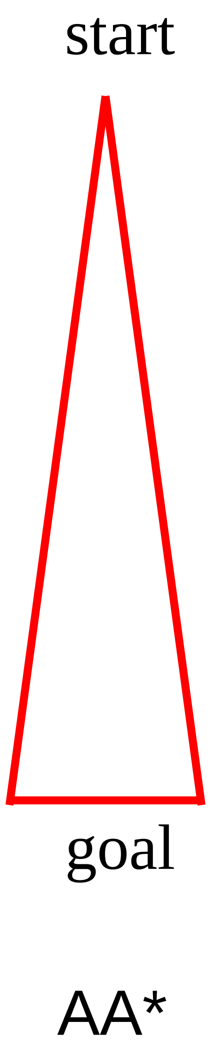


Adaptive A* (AA*)

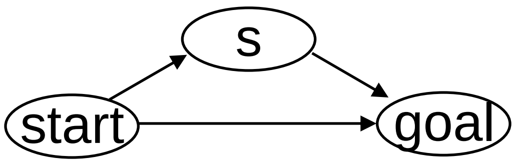
- Adaptive A* (AA*) [Koenig and Likhachev, 2005] speeds up A* searches for a sequence of similar search problems by making the h-values more informed after each search.
- The principle behind AA* was earlier used in Hierarchical A* [Holte et al., 1996].

Adaptive A* (AA*)



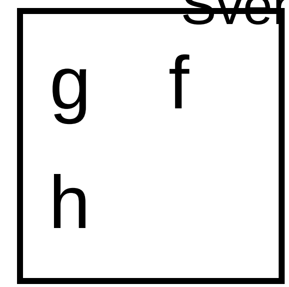


Adaptive A* (AA*)

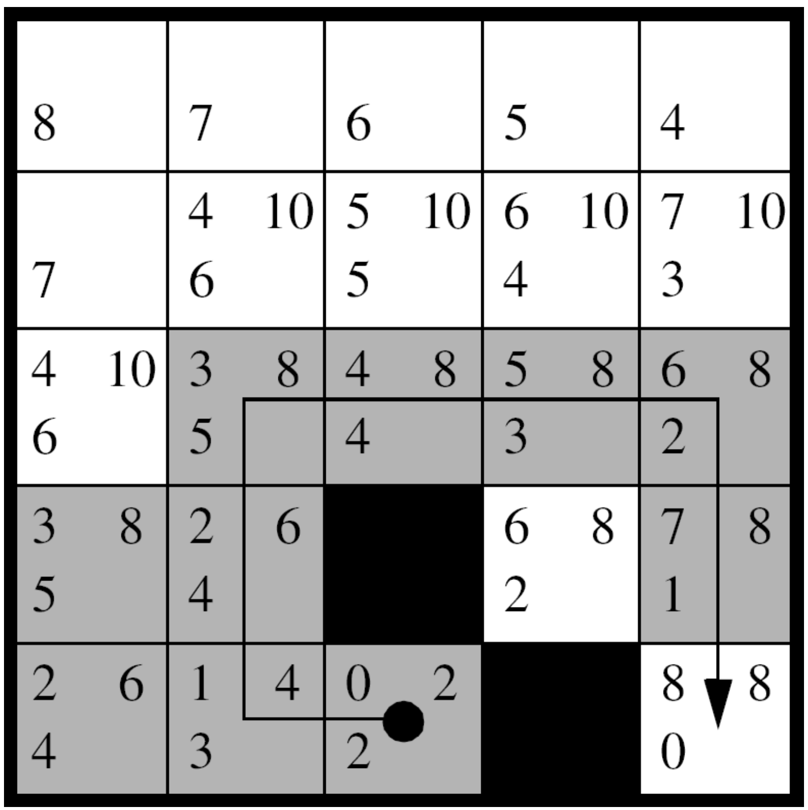


- Consider a state s that was expanded by A* with consistent h-values h_{old}:
  - distance(start,s) + distance(s,goal) ≥ distance(start,goal)
  - distance(s,goal) ≥ distance(start,goal) – distance(start,s)
  - distance(s,goal) ≥ f(goal) – g(s) = h_{new}(s)
- The h-values h_{new} are again consistent.
- The h-values h_{new} dominate the h-values h_{old}.
- These properties continue to hold even if the start changes or the movement costs increase.
- The next A* search with h-values h_{new} expands no more states than an A* search with h-values h_{old} and likely many fewer states.

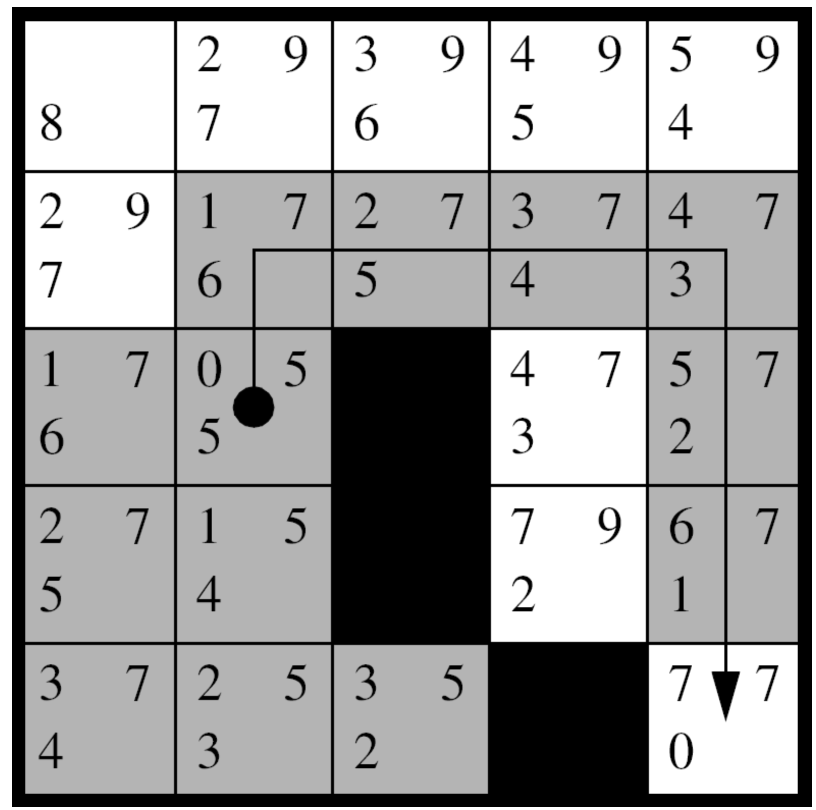
Sven



Adaptive A* (AA*)



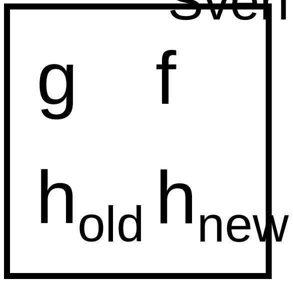
first A* search



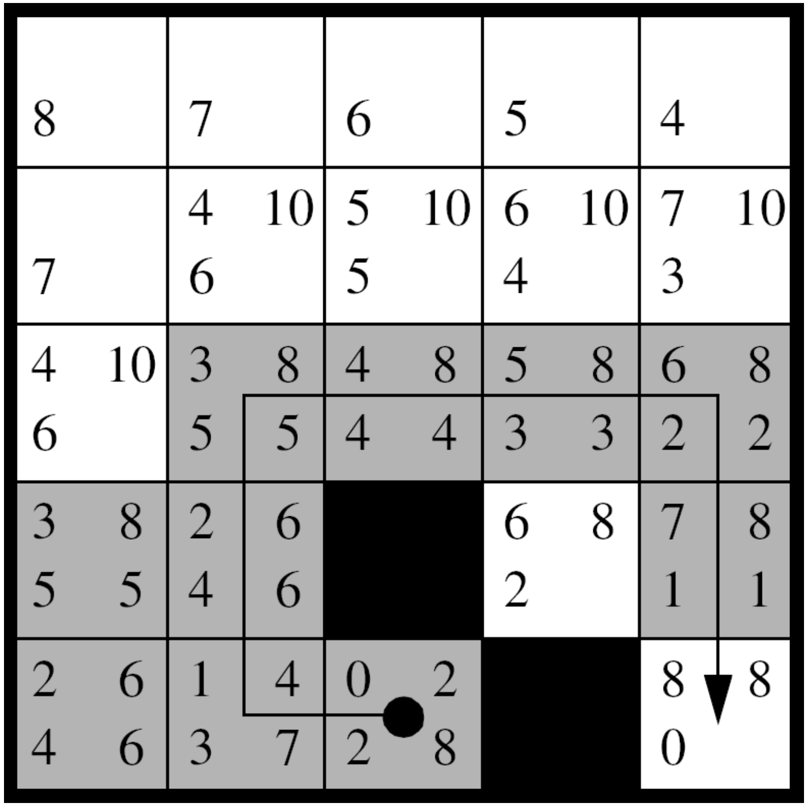
second A* search

4-neighbor grid

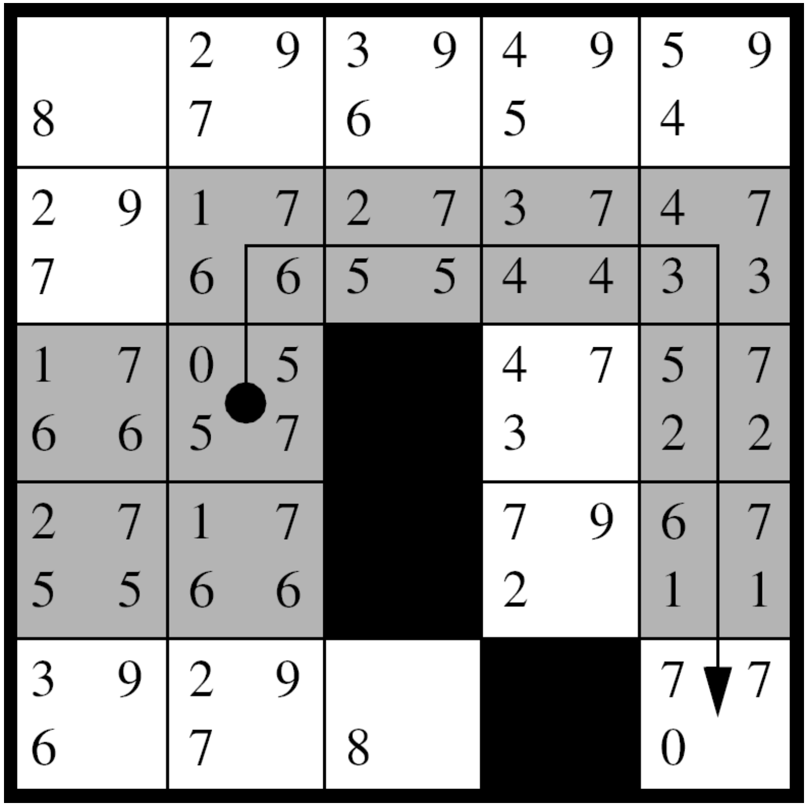
Sven



Adaptive A* (AA*)



first AA* search



second AA* search

4-neighbor grid

Incremental Heuristic Search

- Fringe Saving A* (FSA*)
- Adaptive A* (AA*)
- Lifelong Planning A* (LPA*), D* Lite and Minimax LPA*
- Comparison of D* Lite and Adaptive A*
- Eager and Lazy Moving-Target Adaptive A* (MTAA*)
- Anytime Replanning A* (ARA*)
- Anytime D*

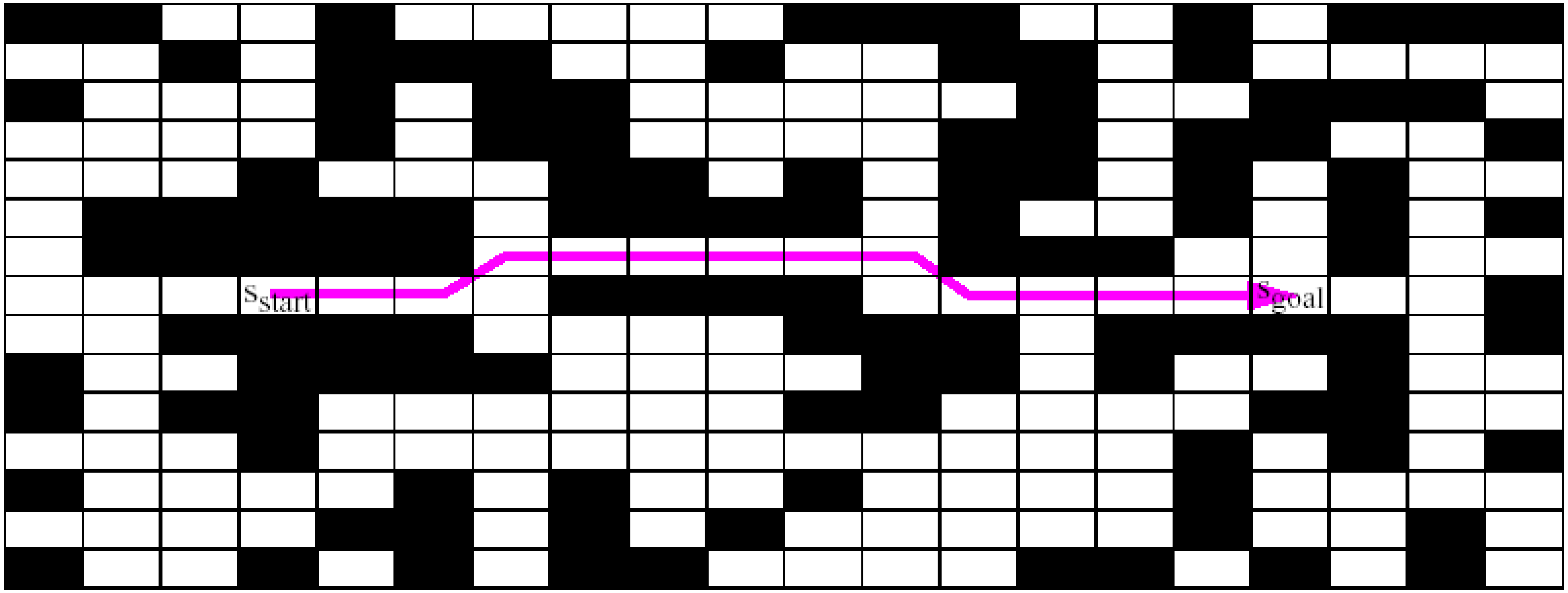
planning time per expansion increases

number of expansions decreases

Lifelong Planning A* (LPA*)

- Lifelong Planning A* (LPA*) [Koenig and Likhachev, 2002] speeds up A* searches for a sequence of similar search problems by recalculating only those g-values in the current search that are important for finding a shortest path **and** have changed from the previous search.
- This can often be understood as transforming the search tree from the previous search to the one of the current search.

Lifelong Planning A* (LPA*)

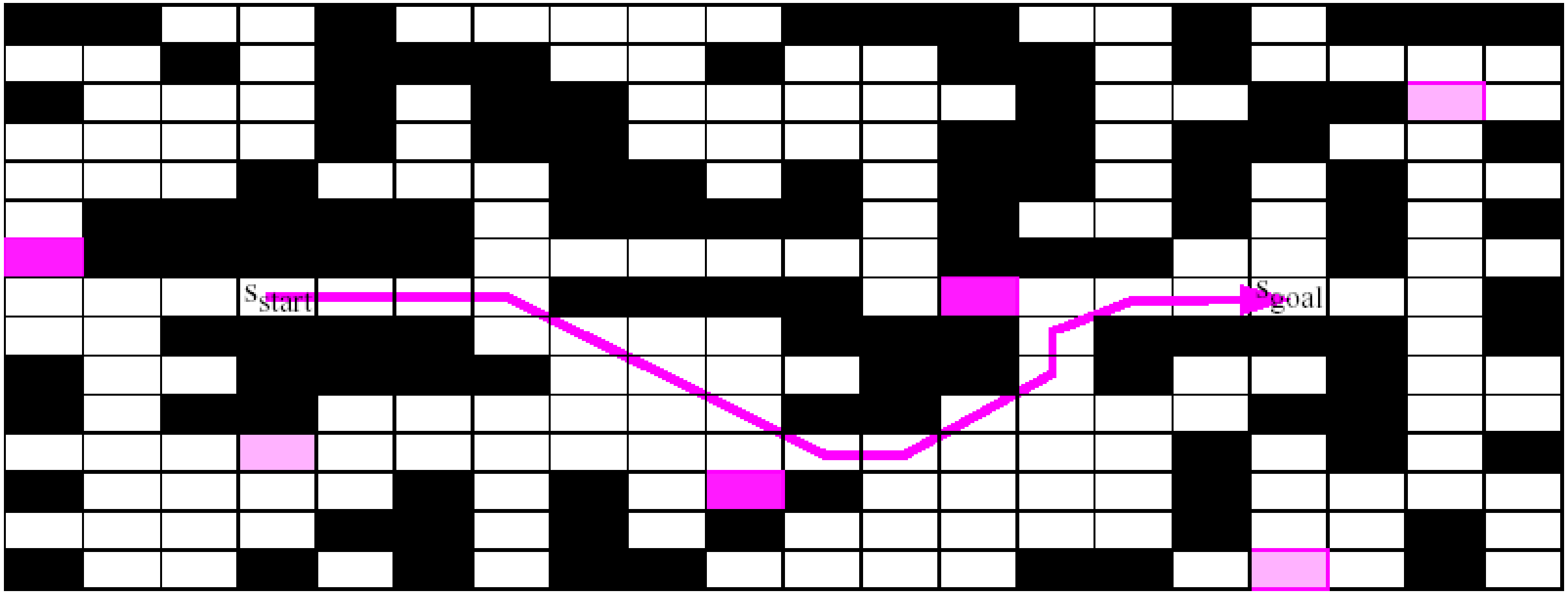


8-neighbor grid

Sven

Lifelong Planning A* (LPA*)

[from slate.com]

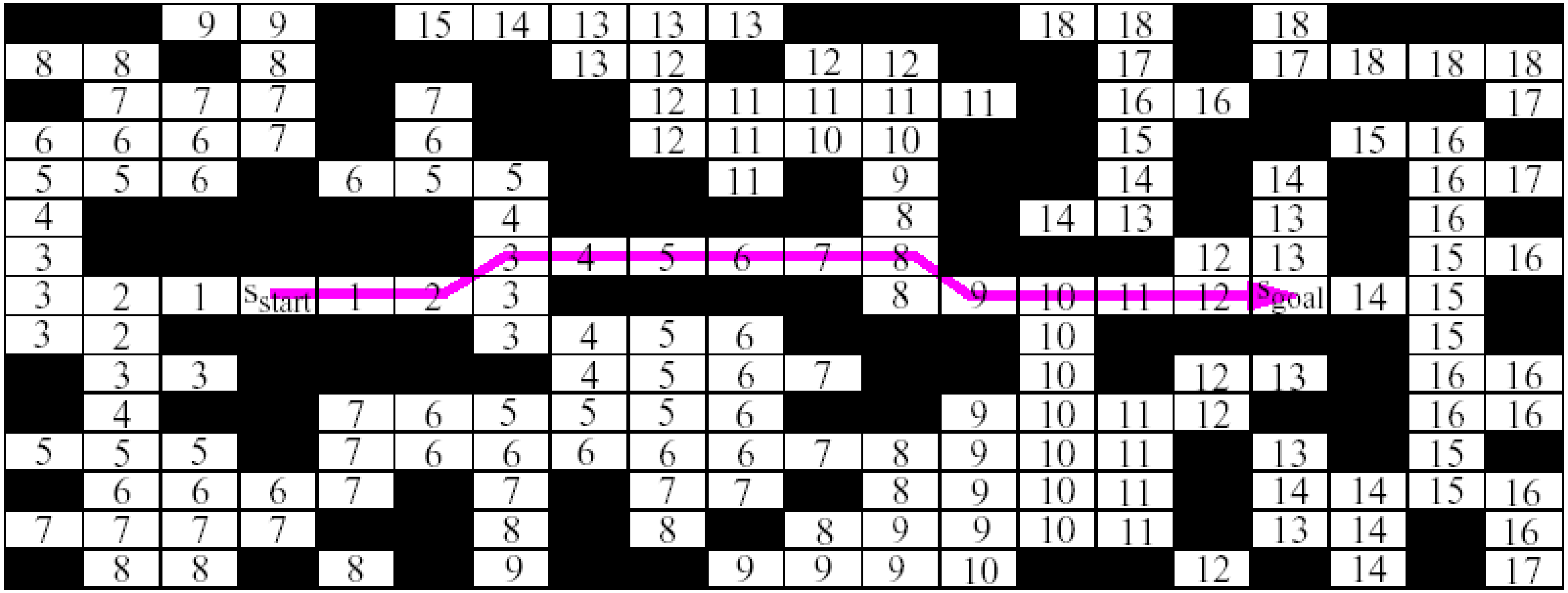


8-neighbor grid

Sven

g

Lifelong Planning A* (LPA*)

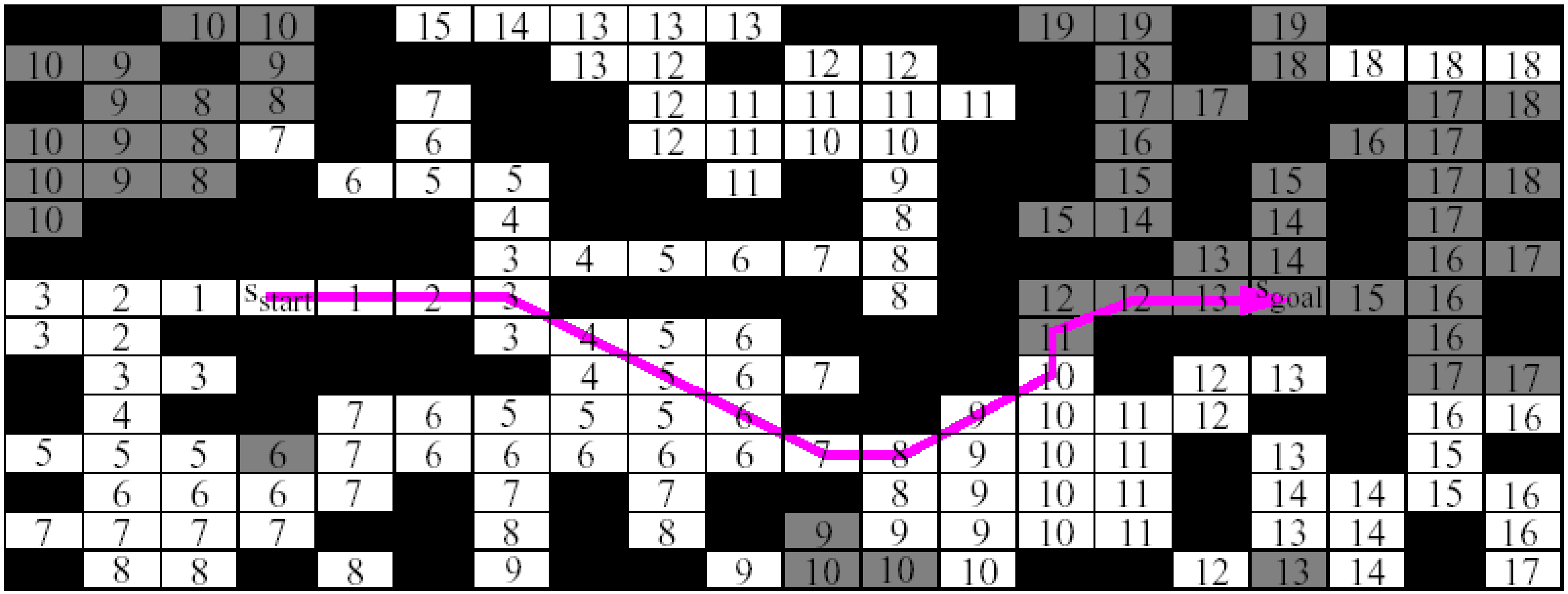


8-neighbor grid

Sven

Lifelong Planning A* (LPA*)

[from slate.com]

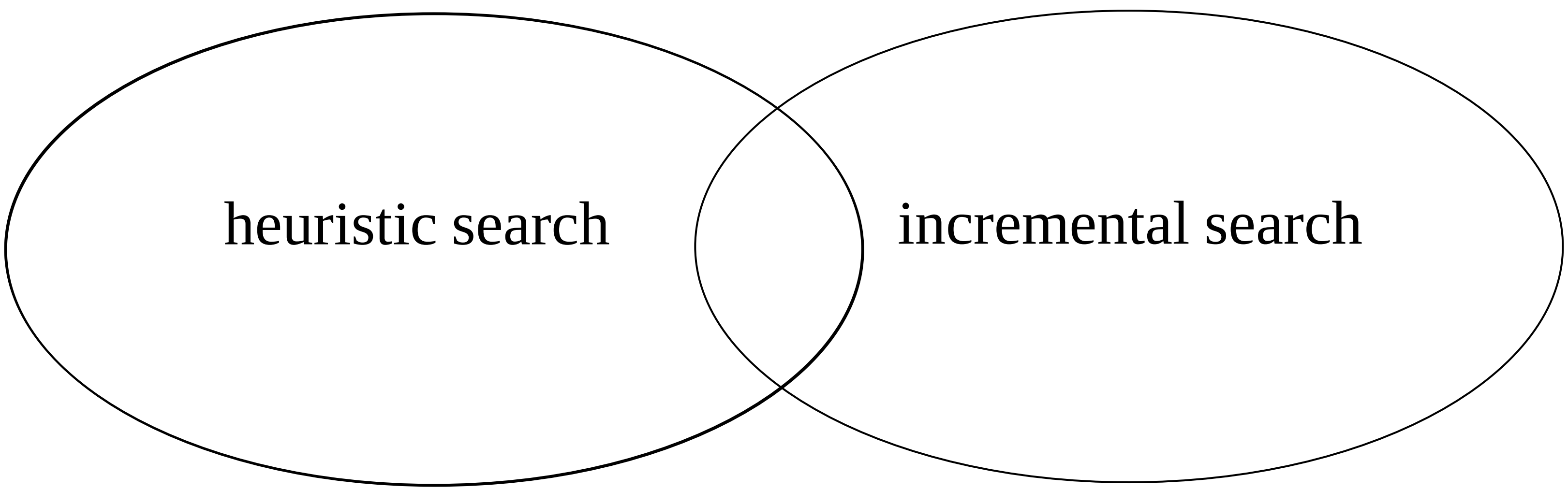


8-neighbor grid

Lifelong Planning A* (LPA*)

artificial intelligence

algorithm theory



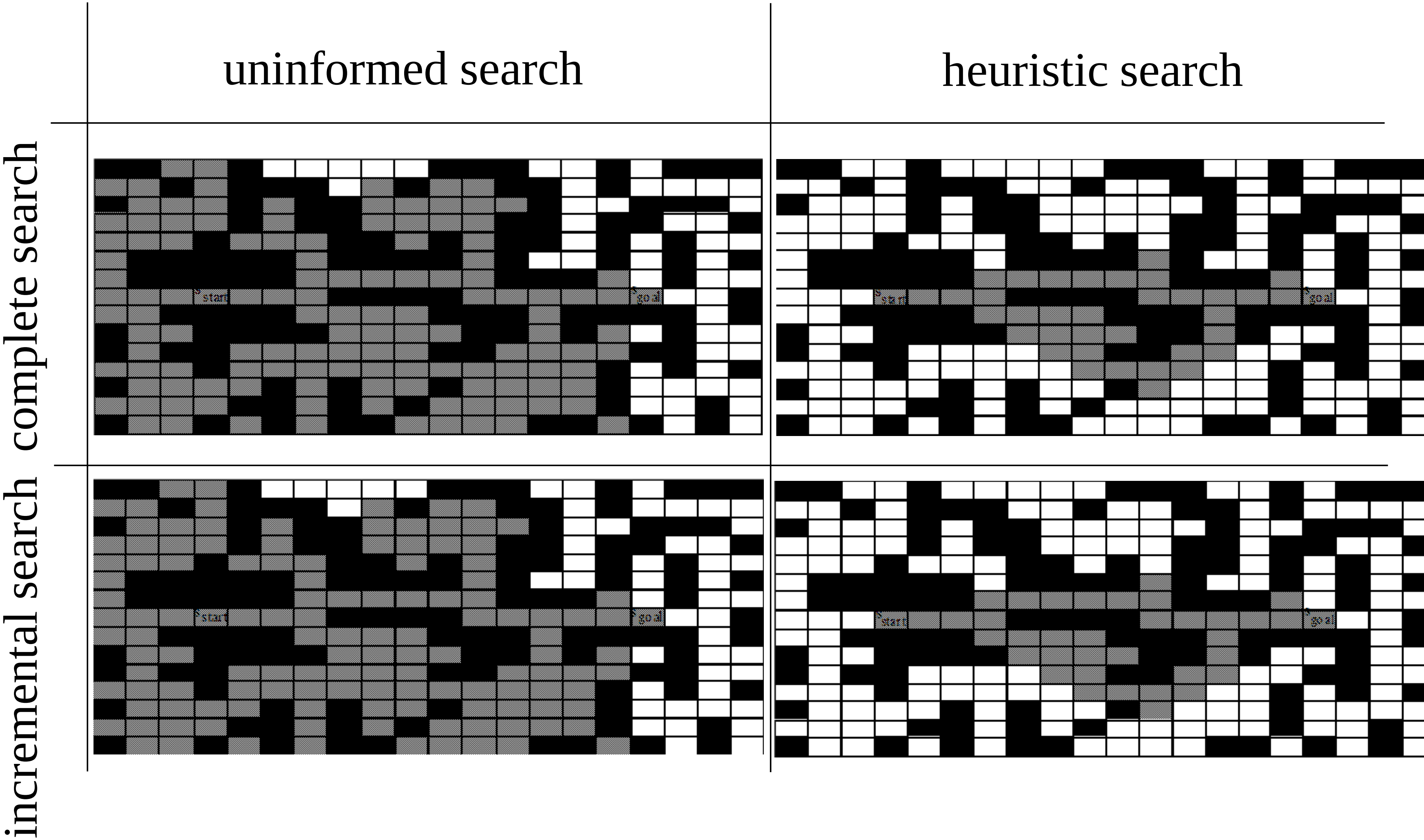
How to search efficiently using h-values to focus the search?

How to search efficiently by reusing information from previous similar searches?

Lifelong Planning A* (LPA*)

	uninformed search	heuristic search
complete search	breadth-first search	A* [Hart, Nilsson, Raphael, 1968]
incremental search	DynamicSWSF-FP with early termination (our addition) [Ramalingam and Reps, 1996]	Lifelong Planning A* (LPA*) [Koenig and Likhachev, 2002]

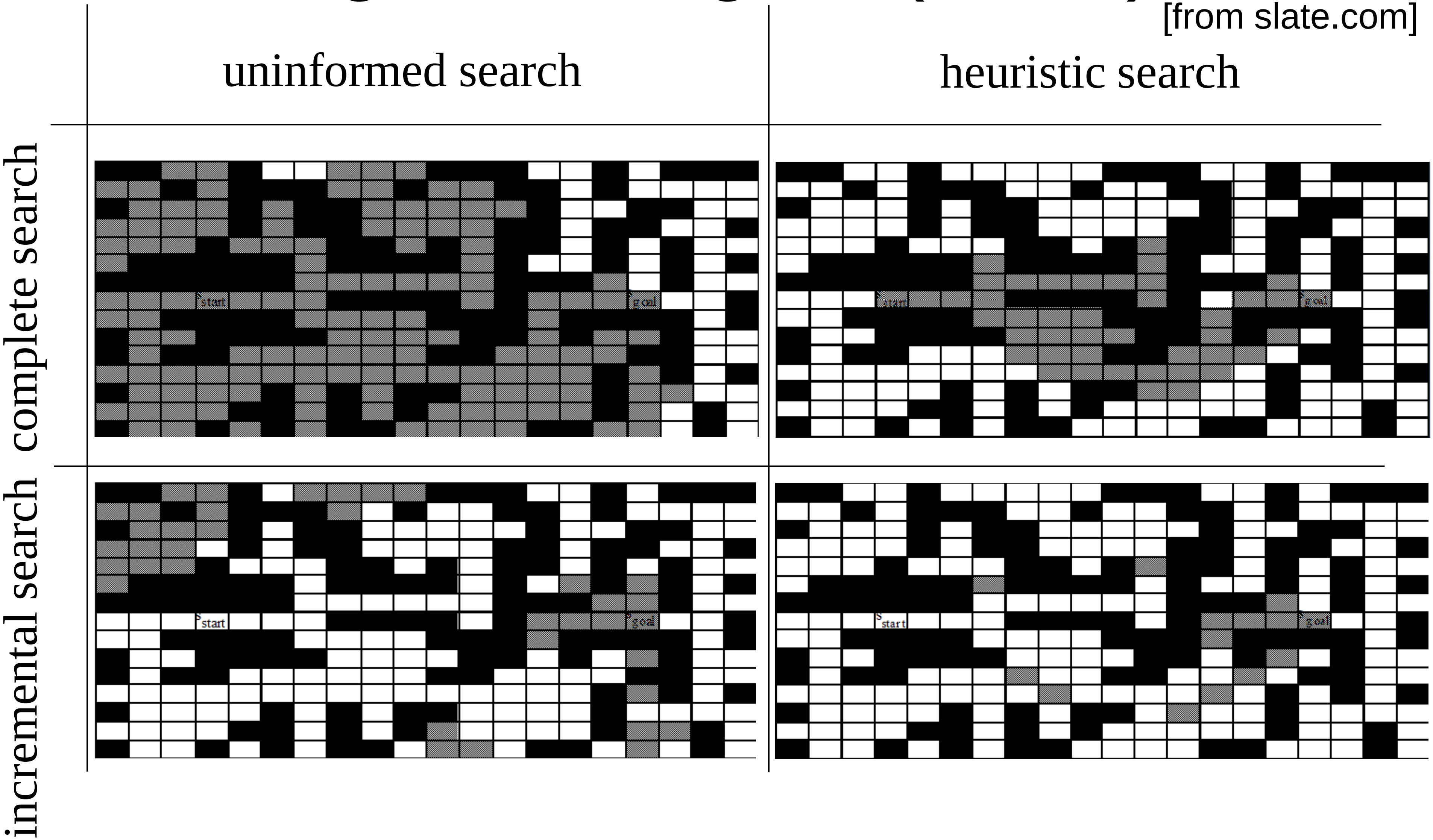
Lifelong Planning A* (LPA*)



Sven

Lifelong Planning A* (LPA*)

[from slate.com]



Lifelong Planning A* (LPA*)

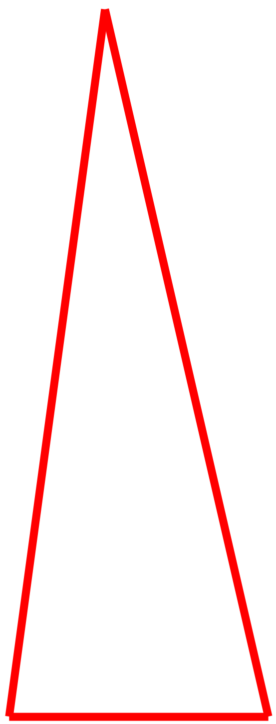
```
procedure CalculateKey(s)
return [min(g(s), rhs(s)) + h(s), min(g(s), rhs(s))];
procedure Initialize()
U := Ø;
for all s ∈ S rhs(s) = g(s) = ∞
rhs(s_start) = 0;
U.Insert(s_start, [h(s_start);0];
procedure UpdateVertex(u)
if (u ≠ s_start) rhs(u) = min_{s' in Pred(u)} (g(s')+c(s',u));
if (u ∈ U) U.Remove(u);
if (g(u) ≠ rhs(u)) U.Insert(u, CalculateKey(u));
procedure ComputeShortestPath()
while (U.TopKey < CalculateKey(s_goal) OR rhs(s_goal) ≠ g(s_goal))
    u = U.Pop();
    if (g(u) > rhs(u))
        g(u) = rhs(u);
        for all s ∈ Succ(u) UpdateVertex(s);
    else
        g(u) = rhs(u);
        for all s ∈ { Succ(u) ∪ u } UpdateVertex(s);
procedure Main()
Initialize();
forever
    ComputeShortestPath();
    Wait for changes in edge costs;
    for all directed edges (u, v) with changed edge costs
        Update the edge cost c(u,v);
        UpdateVertex(v);
```

Lifelong Planning A* (LPA*)

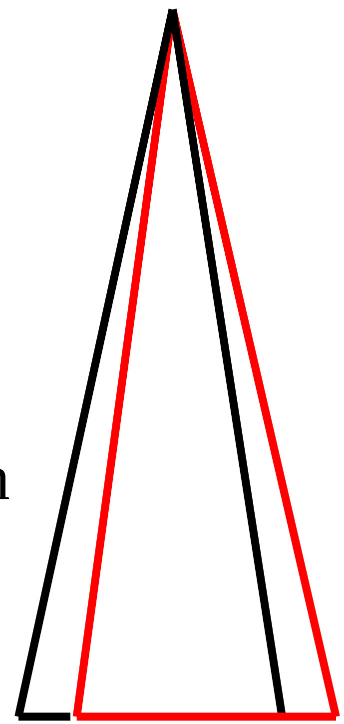
- Grids of size 101 x 101
- Movement costs are one or two with equal probability.

number of movement cost changes	planning time of A*	first planning time of LPA*	replanning time of LPA*	$\frac{\text{replanning time of LPA*}}{\text{planning time of A*}}$
0.2 %	0.299 ms	0.386 ms	0.029 ms	10.4 x
0.4 %	0.336 ms	0.419 ms	0.067 ms	5.0 x
0.6 %	0.362 ms	0.453 ms	0.108 ms	3.3 x
0.8 %	0.406 ms	0.499 ms	0.156 ms	2.6 x
1.0 %	0.370 ms	0.434 ms	0.174 ms	2.1 x

Lifelong Planning A* (LPA*)

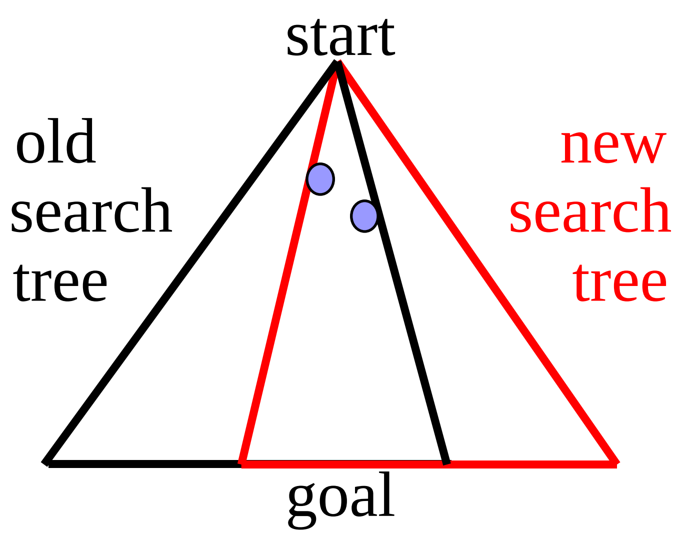


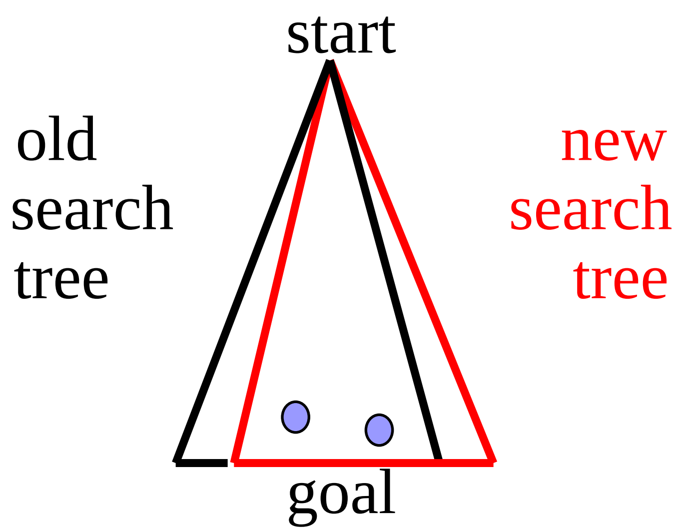
A*



LPA*

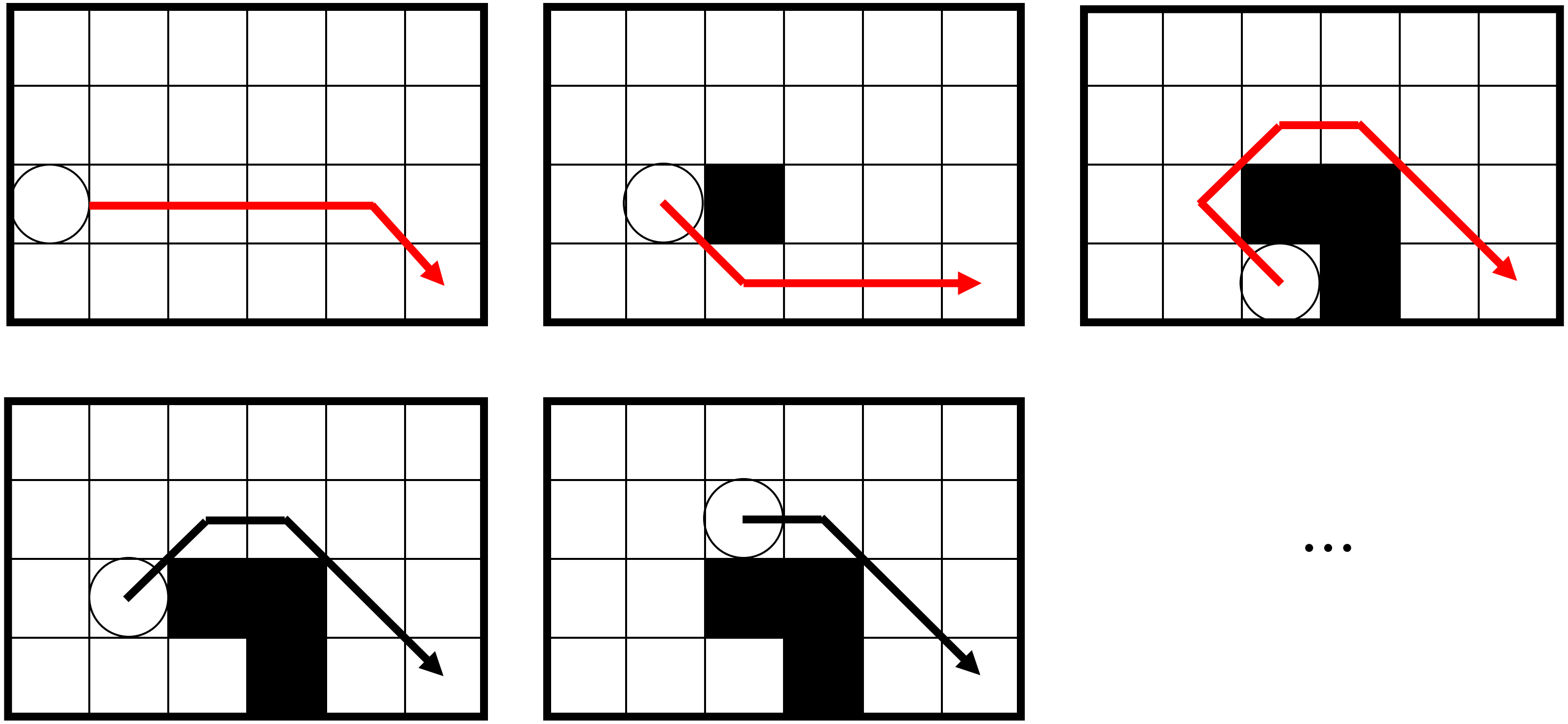
Lifelong Planning A* (LPA*)





- Start of the search must remain unchanged.
- LPA* can expand more states and run slower than A* if
  - the number of changes is large or
  - the changes are close to the start of the search.

Stationary Target

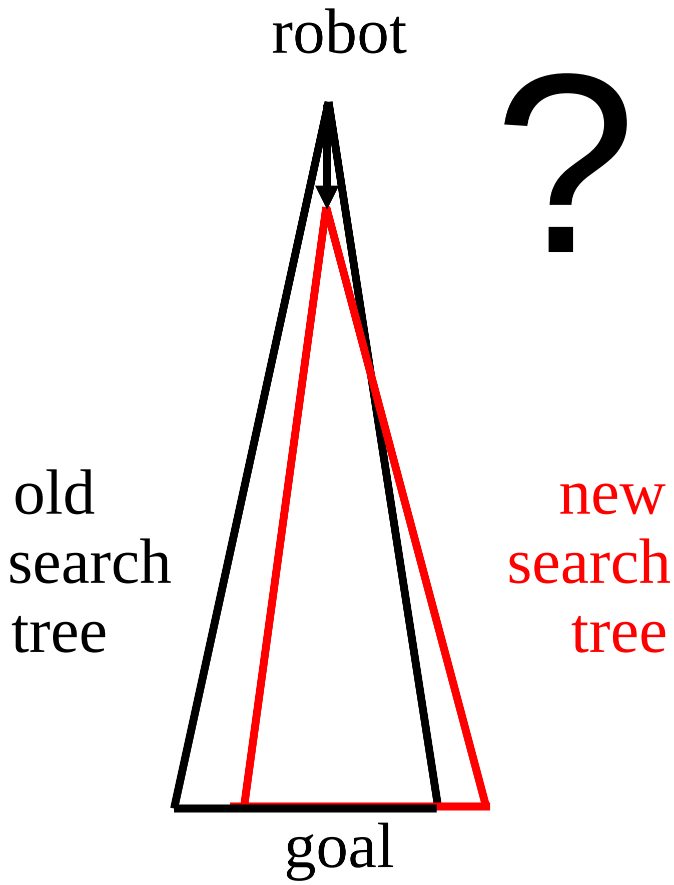


8-neighbor grid

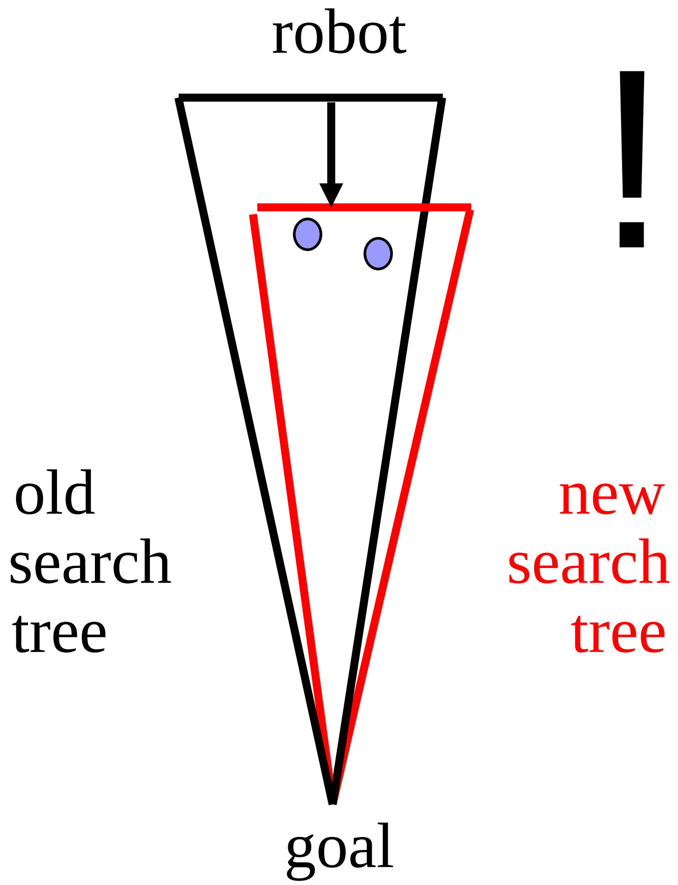
D* Lite

- LPA* needs to search from the destination of the robot to the robot itself because the start of the search needs to remain unchanged.
- LPA* is efficient because the robot observes blockages around itself. Thus, the changes are close to the goal of the search.

D* Lite



LPA*

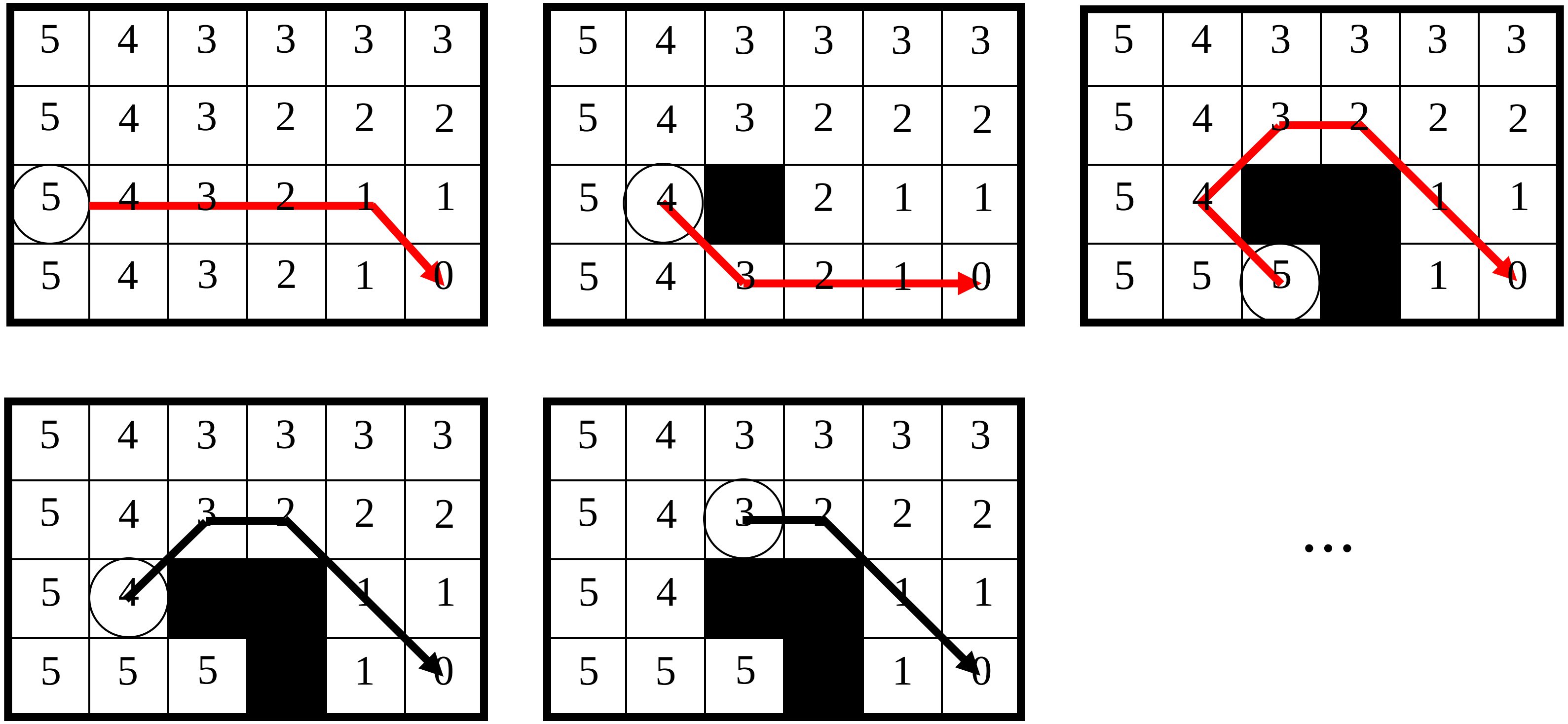


LPA*

Sven

goal
distance

D* Lite



8-neighbor grid

D* Lite

- Random grids of size 129 x 129

	replanning time
uninformed search from scratch	296.0 ms
heuristic search from scratch	10.5 ms
incremental uninformed search	6.1 ms
incremental heuristic search D* [Stentz, 1995] D* was probably the first true incremental heuristic search algorithm, way ahead of its time.	4.2 ms
D* Lite	2.7 ms

speed-up 110x

Incremental Heuristic Search

- Fringe Saving A* (FSA*)
- Adaptive A* (AA*)
- Lifelong Planning A* (LPA*), D* Lite and Minimax LPA*
- Comparison of D* Lite and Adaptive A*
- Eager and Lazy Moving-Target Adaptive A* (MTAA*)
- Anytime Replanning A* (ARA*)
- Anytime D*

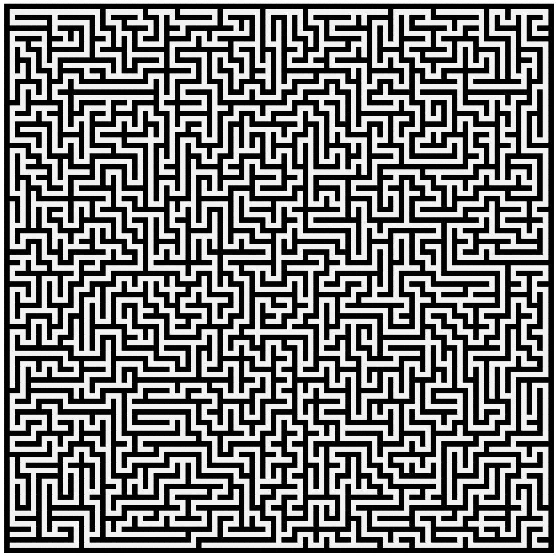
D* Lite vs AA*

LPA*/D* Lite	AA*
■ Adapt previous search tree	■ Improve previous h-values
■ Start of the search must remain unchanged ■ Movement cost in/decreases	■ Goal of the search must remain unchanged ■ Movement cost increases only*
■ Can result in more expansions than A* ■ Fewer expansions on average ■ Slow expansions	■ Guaranteed no more expansions than A* ■ More expansions on average ■ Fast expansions

actually, movement cost in/decreases but AA is more efficient for movement cost increases

D* Lite vs AA*

- Torus-shaped DFS mazes of size 100 x 100



Acyclic mazes generated with DFS

D* Lite vs AA*

	expansions per search	planning time per search
Forward A*	3711	581
Backward A*	4104	644
(Forward) AA*	391	81
(Backward) D* Lite	31	15

Incremental Heuristic Search

- Fringe Saving A* (FSA*)
- Adaptive A* (AA*)
- Lifelong Planning A* (LPA*), D* Lite and Minimax LPA*
- Comparison of D* Lite and Adaptive A*
- Eager and Lazy Moving-Target Adaptive A* (MTAA*)
- Anytime Replanning A* (ARA*)
- Anytime D*

Moving Target

- Moving target search catches a moving target with no a priori given map, always knowing the robot cell.

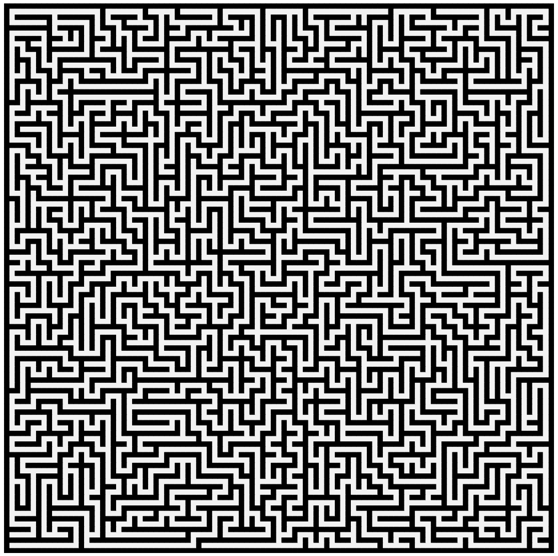
D* Lite vs AA*

LPA*/D* Lite	AA*
■ Adapt previous search tree	■ Improve previous h-values
■ Start of the search must remain unchanged ■ Movement cost in/decreases	■ Goal of the search must remain unchanged ■ Movements cost increases only*
■ Can result in more expansions than A* ■ Fewer expansions on average ■ Slow expansions	■ Guaranteed no more expansions than A* ■ More expansions on average ■ Fast expansions

actually, movement cost in/decreases but AA is more efficient for movement cost increases

D* Lite vs MTAA*

- Torus-shaped DFS mazes of size 100 x 100
- Randomly moving target that pauses every 10th move



Acyclic mazes generated with DFS

D* Lite vs MTAA*

	expansions per search	planning time per search
Forward A*	3703	570
Backward A*	4519	722
Forward Lazy MTAA*	2334	465
Backward Lazy MTAA*	2025	411
Agent-Centric D* Lite	2229	1481
Target-Centric D* Lite	806	833

Incremental Heuristic Search

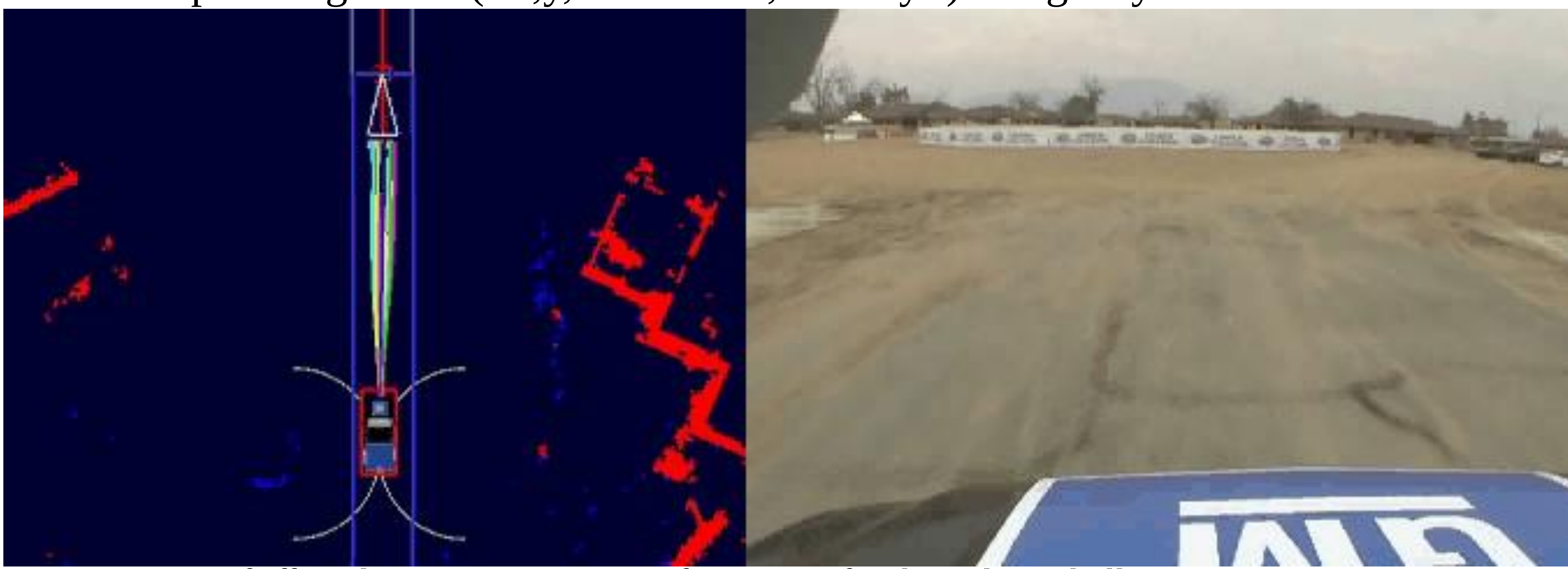
- Fringe Saving A* (FSA*)
- Adaptive A* (AA*)
- Lifelong Planning A* (LPA*), D* Lite and Minimax LPA*
- Comparison of D* Lite and Adaptive A*
- Eager and Lazy Moving-Target Adaptive A* (MTAA*)
- Anytime Replanning A* (ARA*)
- Anytime D*

Dynamic and Partially-known Environments

Maxim

- Planning in
  - partially-known environments is a repeated process
  - dynamic environments is also a repeated process

planning in 4D (<x,y,orientation,velocity>) using Anytime D*

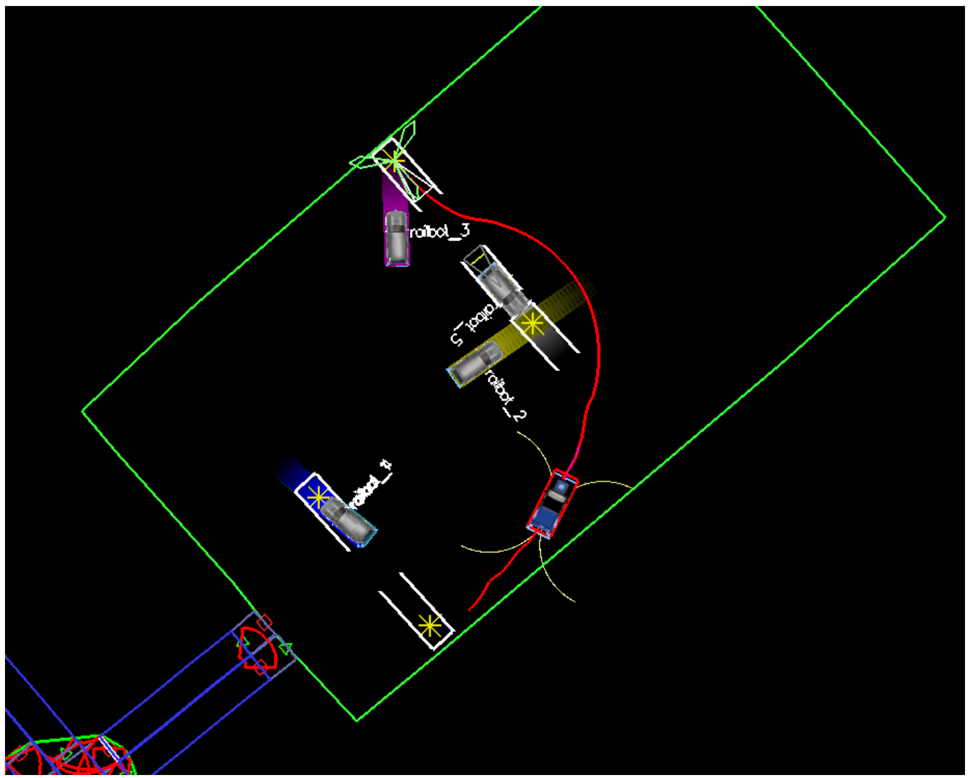


part of efforts by Tartanracing team from CMU for the Urban Challenge 2007 race

Dynamic and Partially-known Environments

- Planning in
  - partially-known environments is a repeated process
  - dynamic environments is also a repeated process

planning in dynamic environments



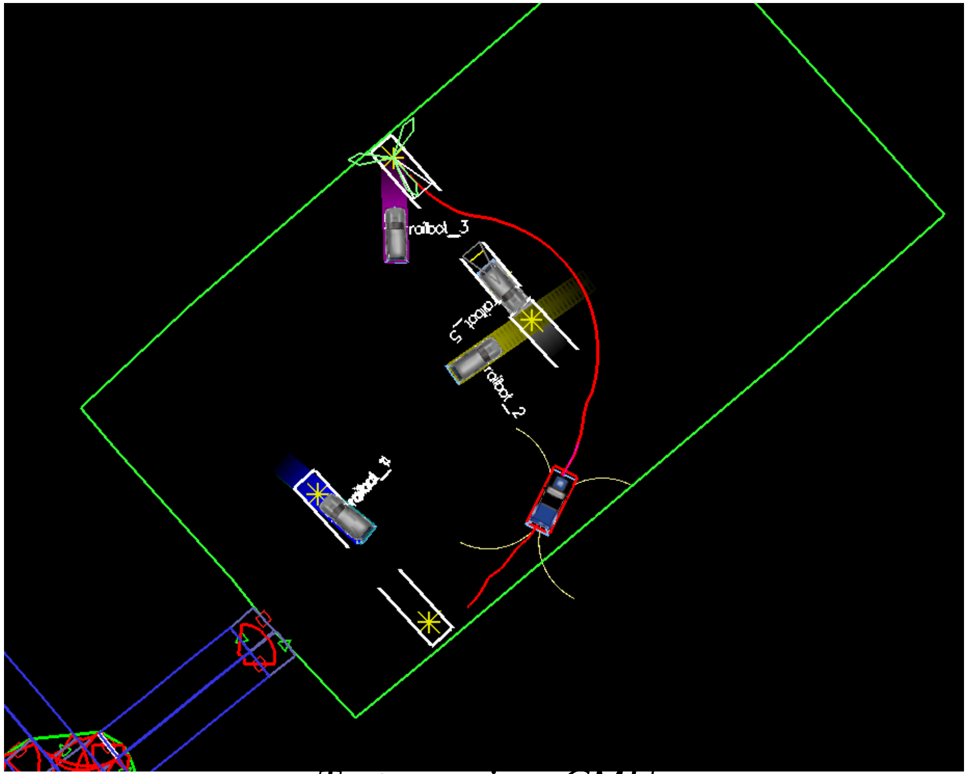
Tartanracing, CMU

Maxim

Dynamic and Partially-known Environments

- Need to re-plan fast!
- Two ways to help with this requirement
  – anytime planning – return the best plan possible within T msecs
  – incremental planning – reuse previous planning efforts

planning in dynamic environments



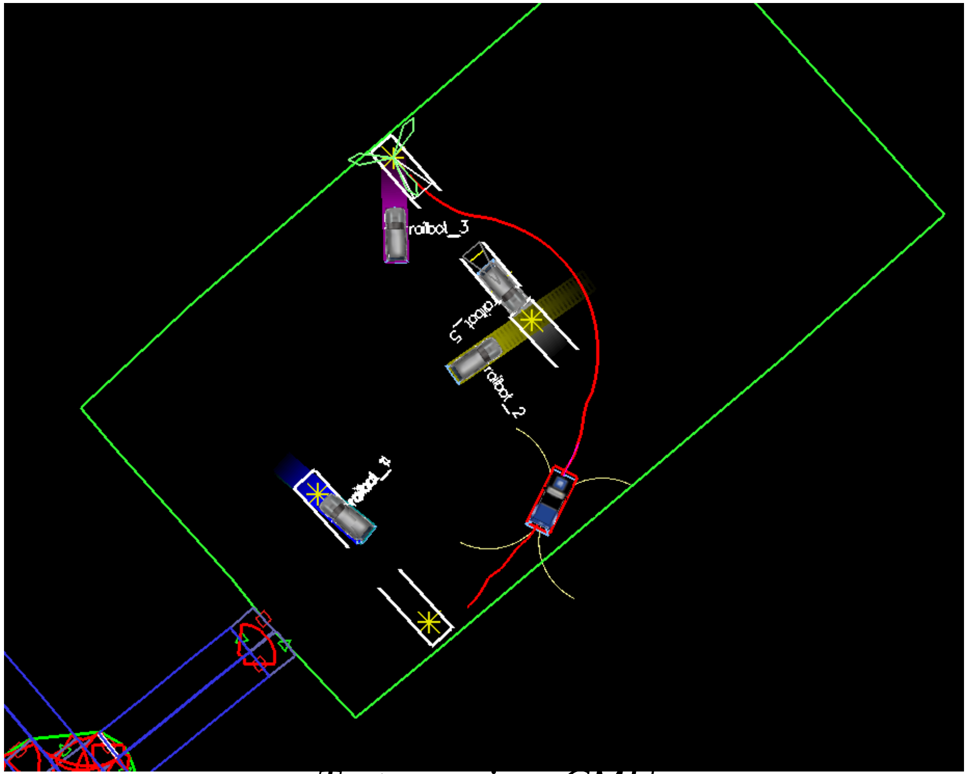
Tartanracing, CMU

Dynamic and Partially-known Environments

Maxim

- Need to re-plan fast!
- Two ways to help with this requirement
  - anytime planning – return the best plan possible within T msecs
  - incremental planning – reuse previous planning efforts

planning in dynamic environments



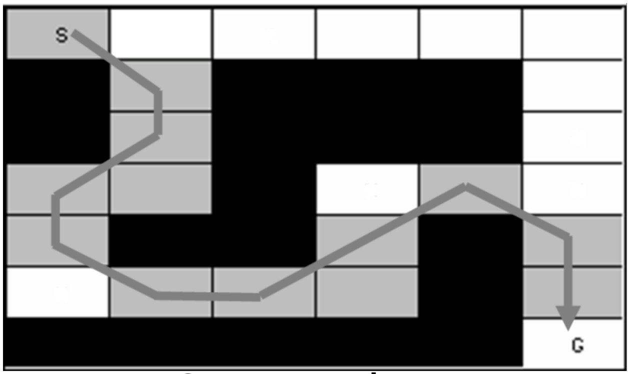
Tartanracing, CMU

Anytime Search based on weighted A*

Maxim

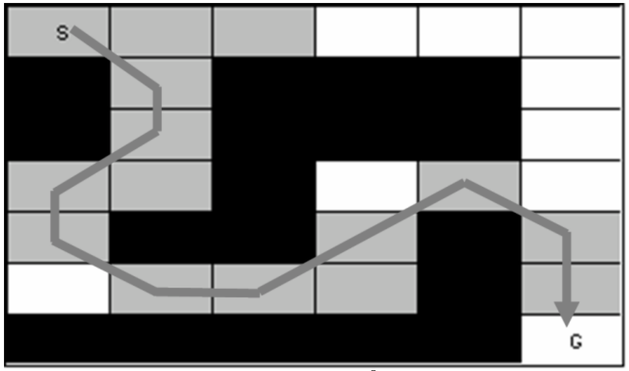
- Constructing anytime search based on weighted A*:
  - find the best path possible given some amount of time for planning
  - do it by running a series of weighted A* searches with decreasing ε:

ε =2.5



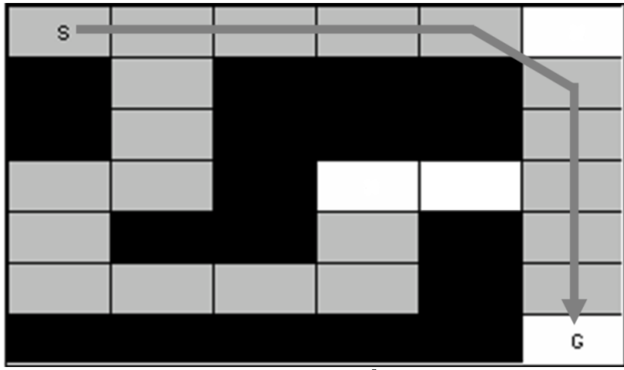
13 expansions
solution=11 moves

ε =1.5



15 expansions
solution=11 moves

ε =1.0



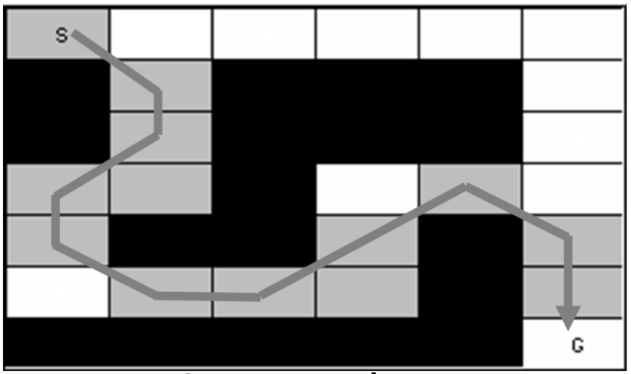
20 expansions
solution=10 moves

Anytime Search based on weighted A*

Maxim

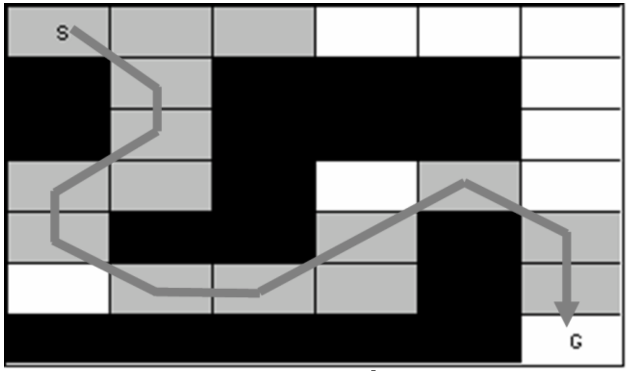
- Constructing anytime search based on weighted A*:
  - find the best path possible given some amount of time for planning
  - do it by running a series of weighted A* searches with decreasing ε:

$\varepsilon = 2.5$



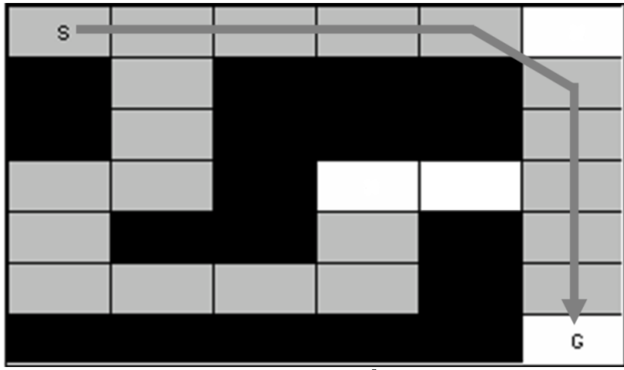
13 expansions
solution=11 moves

$\varepsilon = 1.5$



15 expansions
solution=11 moves

$\varepsilon = 1.0$



20 expansions
solution=10 moves

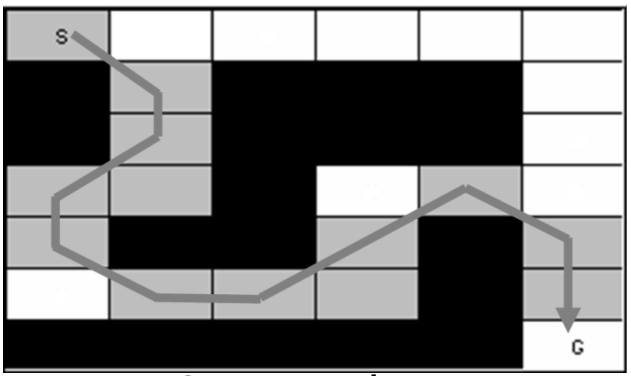
- Inefficient because
  - many state values remain the same between search iterations
  - we should be able to reuse the results of previous searches

Anytime Search based on weighted A*

Maxim

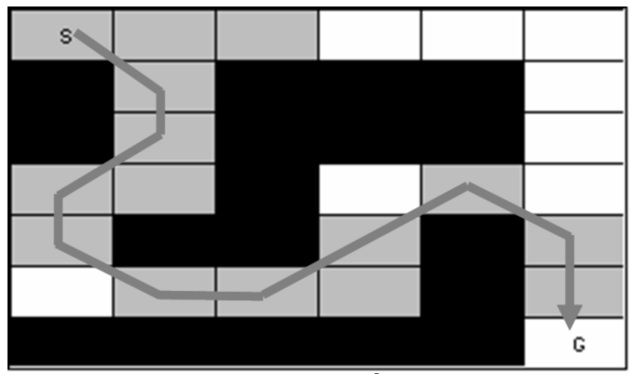
- Constructing anytime search based on weighted A*:
  - find the best path possible given some amount of time for planning
  - do it by running a series of weighted A* searches with decreasing ε:

ε =2.5



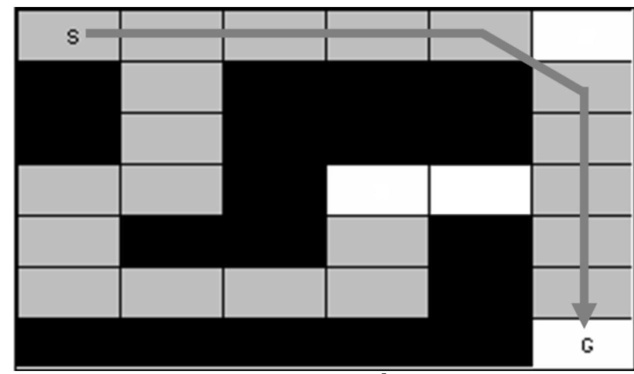
13 expansions
solution=11 moves

ε =1.5



15 expansions
solution=11 moves

ε =1.0



20 expansions
solution=10 moves

- ARA*
  - an efficient version of the above that reuses state values within any search iteration

A* with Reuse of State Values

- Alternative view of A* [Likhachev et. al., AIJ'08]:
  - basis for efficient reuse of search efforts in ARA*/LPA*/D* Lite and their extensions
  - simple but useful trick

A* with Reuse of State Values

Maxim

- Alternative view of A*

all v-values initially are infinite;

ComputePath function

while($f(s_{goal})$ > minimum f-value in *OPEN*)

remove s with the smallest $[g(s)+ h(s)]$ from *OPEN*;

insert s into *CLOSED*;

$v(s)=g(s);$

for every successor s' of s

if $g(s') > g(s) + c(s,s')$

$g(s') = g(s) + c(s,s');$

insert s' into *OPEN*;

A* with Reuse of State Values

- Alternative view of A*

all v-values initially are infinite;

v-value – the value of a state during its expansion (infinite if state was never expanded)

ComputePath function

while($f(s_{goal})$ > minimum f-value in *OPEN*)
  remove s with the smallest $[g(s)+h(s)]$ from *OPEN*;
  insert s into *CLOSED*;
  $v(s)=g(s);$
  for every successor s' of s
    if $g(s') > g(s) + c(s,s')$
      $g(s') = g(s) + c(s,s');$
      insert s' into *OPEN*;

A* with Reuse of State Values

- Alternative view of A*

all *v*-values initially are infinite;

ComputePath function

while($f(s_{goal})$ > minimum f-value in *OPEN*)

remove s with the smallest $[g(s)+ h(s)]$ from *OPEN*;

insert s into *CLOSED*;

$v(s)=g(s);$

for every successor s' of s

if $g(s') > g(s) + c(s,s')$

$g(s') = g(s) + c(s,s');$

insert s' into *OPEN*;

- $g(s') = \min_{s'' \in pred(s')} v(s'') + c(s'',s')$

A* with Reuse of State Values

- Alternative view of A*

all v-values initially are infinite;

ComputePath function

while($f(s_{goal})$ > minimum f-value in *OPEN*)

remove s with the smallest $[g(s)+ h(s)]$ from *OPEN*;

insert s into *CLOSED*;

$v(s)=g(s);$

for every successor s' of s

if $g(s') > g(s) + c(s,s')$

$g(s') = g(s) + c(s,s');$

insert s' into *OPEN*;

overconsistent state

consistent state

- $g(s') = \min_{s'' \in pred(s')} v(s'') + c(s'',s')$
- *OPEN:* a set of states with $v(s) > g(s)$

  all other states have $v(s) = g(s)$

A* with Reuse of State Values

Maxim

- Alternative view of A*

all *v*-values initially are infinite;

ComputePath function

while($f(s_{goal})$ > minimum f-value in *OPEN*)

remove *s* with the smallest $[g(s)+ h(s)]$ from *OPEN*;

insert *s* into *CLOSED*;

$v(s)=g(s);$

for every successor *s'* of *s*

if $g(s') > g(s) + c(s,s')$

$g(s') = g(s) + c(s,s');$

insert *s'* into *OPEN*;

- $g(s') = \min_{s'' \in pred(s')} v(s'') + c(s'',s')$
- *OPEN:* a set of states with $v(s) > g(s)$
  all other states have $v(s) = g(s)$
- this A* expands overconsistent states in the order of their f-values

A* with Reuse of State Values

- Making A* reuse old values:

initialize *OPEN* with all overconsistent states;

ComputePathwithReuse function

while($f(s_{goal})$ > minimum f-value in *OPEN*)

  remove s with the smallest $[g(s)+h(s)]$ from *OPEN*;

  insert s into *CLOSED*;

  $v(s)=g(s);$

  for every successor s' of s

    if $g(s') > g(s) + c(s,s')$

      $g(s') = g(s) + c(s,s');$

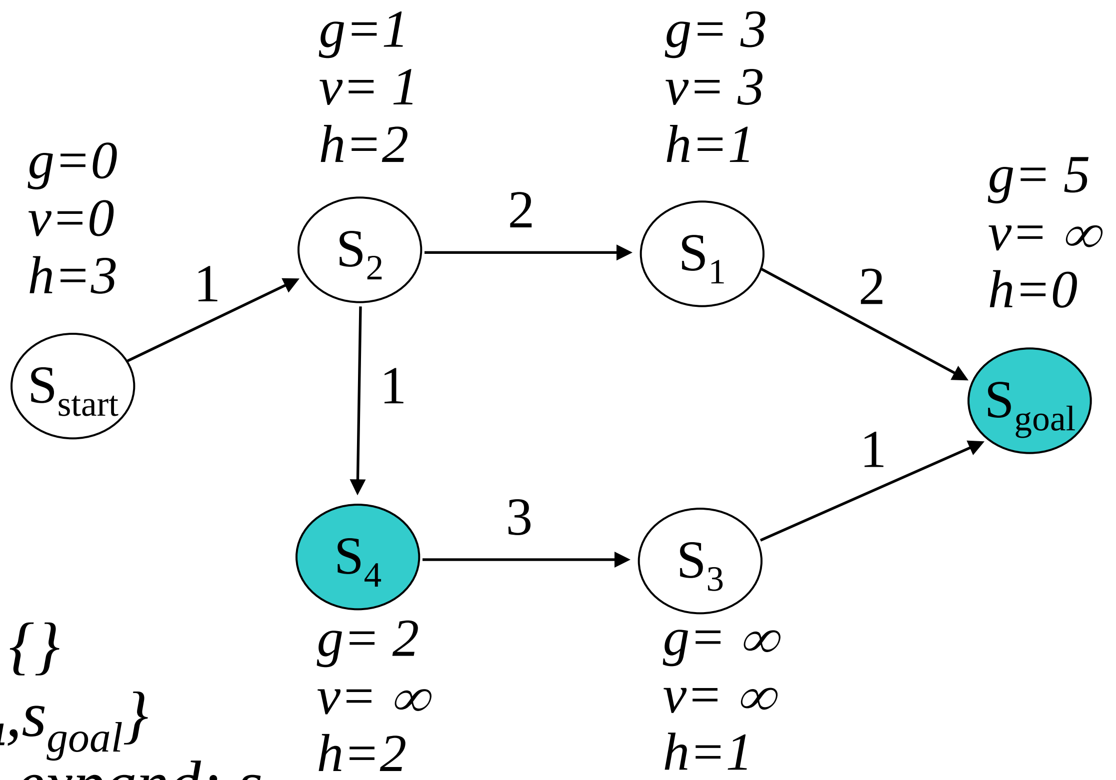
      insert s' into *OPEN*;

all you need to do to make it reuse old values!

- $g(s') = \min_{s'' \in pred(s')} v(s'') + c(s'',s')$
- *OPEN:* a set of states with $v(s) > g(s)$
  all other states have $v(s) = g(s)$
- this A* expands overconsistent states in the order of their f-values

A* with Reuse of State Values

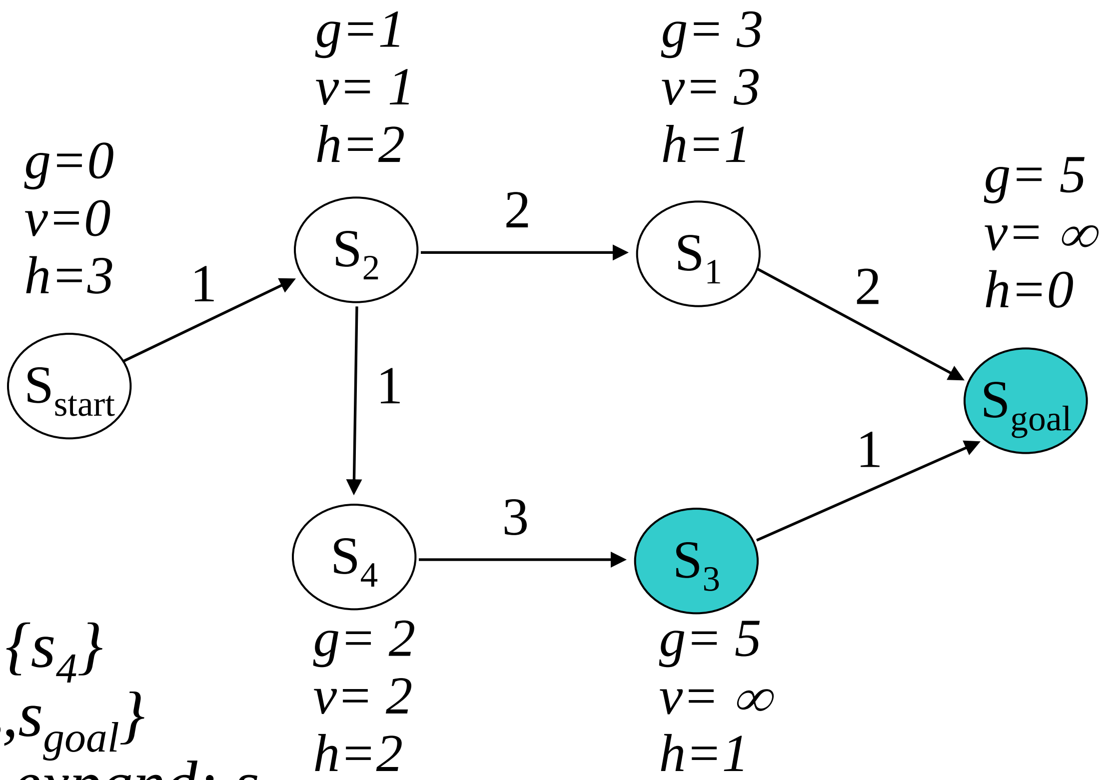
Maxim



CLOSED = {}
OPEN = {s_4,s_{goal}}
next state to expand: s_4

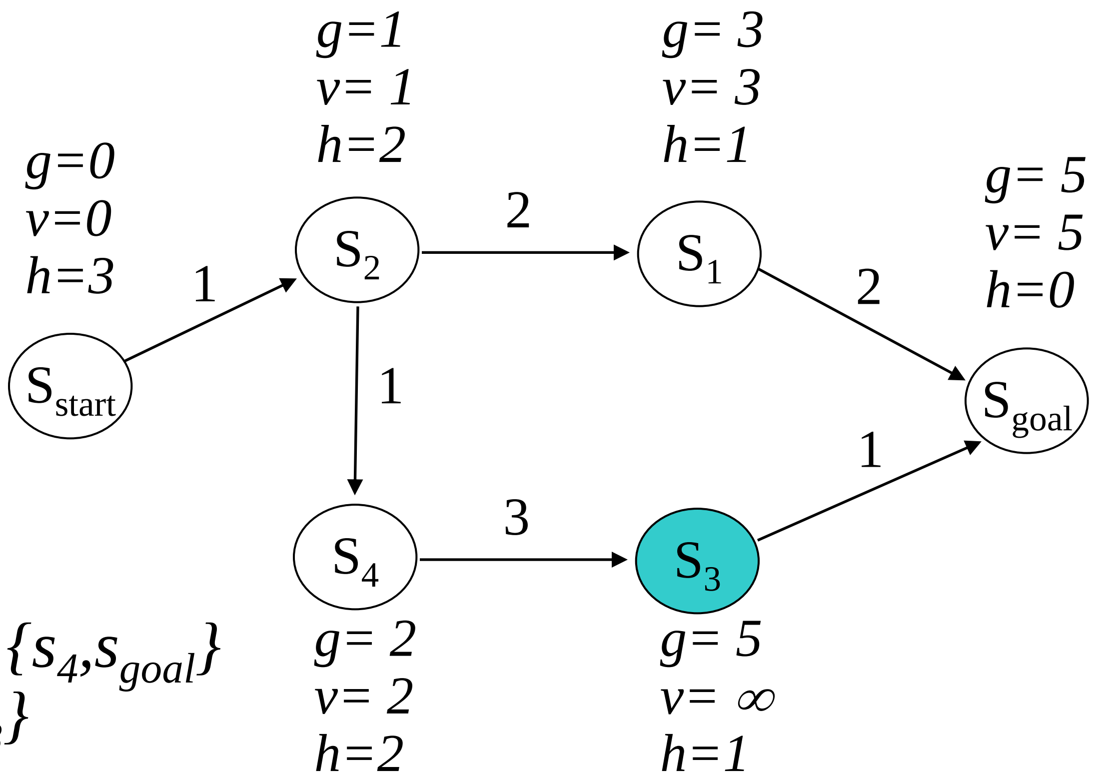
$g(s') = \min_{s'' \in pred(s')} v(s'') + c(s'',s')$
initially OPEN contains all overconsistent states

A* with Reuse of State Values



CLOSED = $\{s_4\}$
OPEN = $\{s_3, s_{goal}\}$
next state to expand: s_{goal}

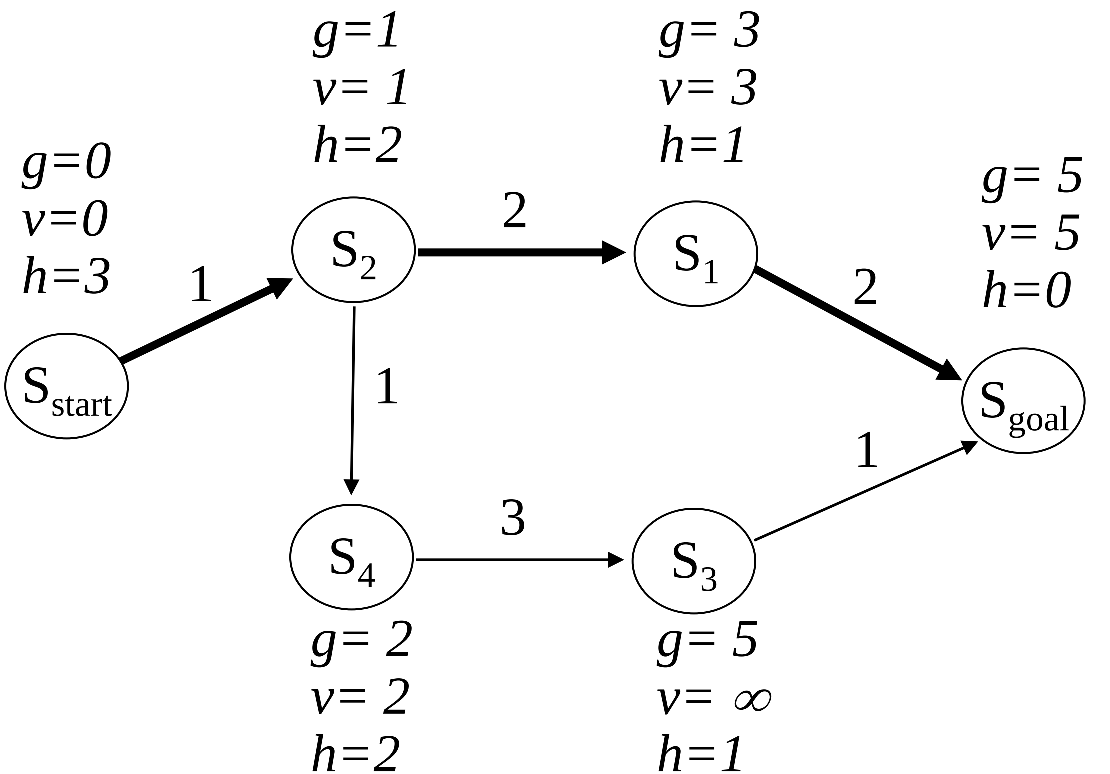
A* with Reuse of State Values



CLOSED = $\{s_4, s_{goal}\}$
OPEN = $\{s_3\}$
done

after ComputePathwithReuse terminates:
all g-values of states are equal to final A g-values*

A* with Reuse of State Values



we can now compute a least-cost path

A* with Reuse of State Values

- Making weighted A* reuse old values:

initialize *OPEN* with all overconsistent states;

ComputePathwithReuse function

while($f(s_{goal})$ > minimum f-value in *OPEN*)

remove s with the smallest $[g(s)+ \varepsilon h(s)]$ from *OPEN*;

insert s into *CLOSED*;

$v(s)=g(s);$

for every successor s' of s

if $g(s') > g(s) + c(s,s')$

$g(s') = g(s) + c(s,s');$

if s' not in *CLOSED* then insert s' into *OPEN*;

*the exact same thing as with A**

just make sure no state is expanded multiple times

Anytime Repairing A* (ARA*)

Maxim

- Efficient series of weighted A* searches with decreasing ε:

set ε to large value;
$g(s_{start}) = 0$; v-values of all states are set to infinity; $OPEN = \{s_{start}\}$;
while $\varepsilon \geq 1$
    $CLOSED = \{\}$;
    ComputePathwithReuse();
    publish current ε suboptimal solution;
    decrease ε;
    initialize $OPEN$ with all overconsistent states;

ARA*

Maxim

- Efficient series of weighted A* searches with decreasing ε:

set ε to large value;

$g(s_{start}) = 0$; v-values of all states are set to infinity; $OPEN = \{s_{start}\}$;

while $\varepsilon \geq 1$

    $CLOSED = \{\}$;

    ComputePathwithReuse();

    publish current ε suboptimal solution;

    decrease ε;

    initialize $OPEN$ with all overconsistent states;

need to keep track of those

ARA*

- Efficient series of weighted A* searches with decreasing ε:

initialize *OPEN* with all overconsistent states;

ComputePathwithReuse function

while($f(s_{goal})$ > minimum f-value in *OPEN*)

remove s with the smallest $[g(s)+ \varepsilon h(s)]$ from *OPEN*;

insert s into *CLOSED*;

$v(s)=g(s);$

for every successor s' of s

if $g(s') > g(s) + c(s,s')$

$g(s') = g(s) + c(s,s');$

if s' not in *CLOSED* then insert s' into *OPEN*;

otherwise insert s' into *INCONS*

- *OPEN U INCONS* = all overconsistent states

ARA*

- Efficient series of weighted A* searches with decreasing ε:

set ε to large value;

$g(s_{start}) = 0$; v-values of all states are set to infinity; $OPEN = \{s_{start}\}$;

while $\varepsilon \geq 1$

    $CLOSED = \{\}$; $INCONS = \{\}$;

    ComputePathwithReuse();

    publish current ε suboptimal solution;

    decrease ε;

    initialize $OPEN = OPEN \cup INCONS$;

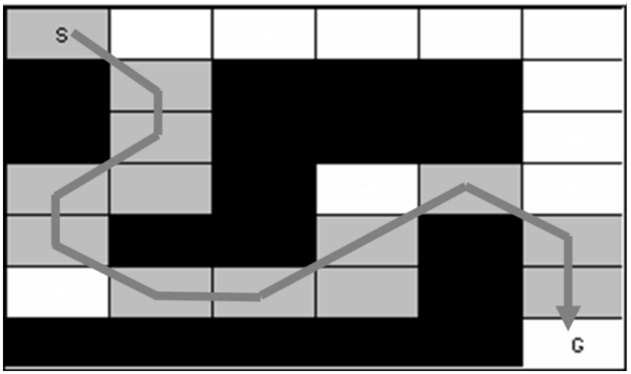
all overconsistent states (exactly what we need!)

Maxim

ARA*

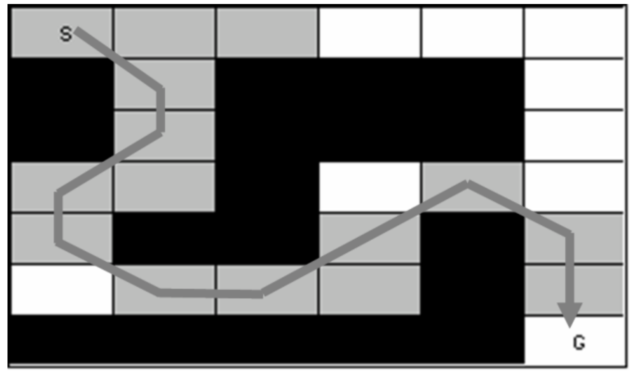
- **A series of weighted A* searches**

ε =2.5



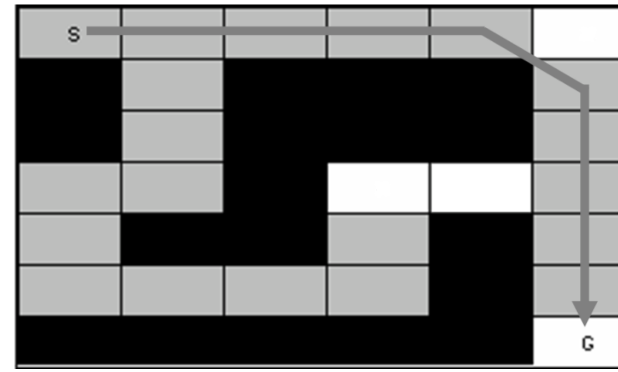
13 expansions
solution=11 moves

ε =1.5



15 expansions
solution=11 moves

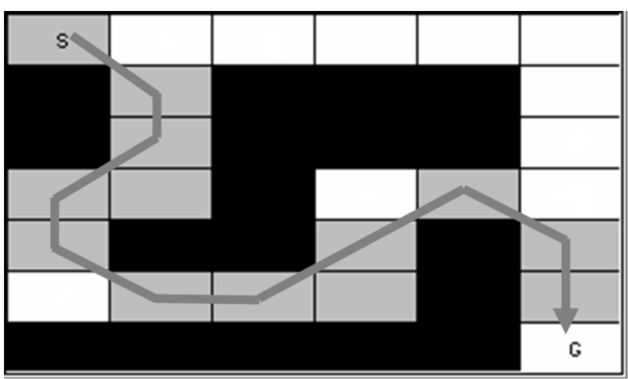
ε =1.0



20 expansions
solution=10 moves

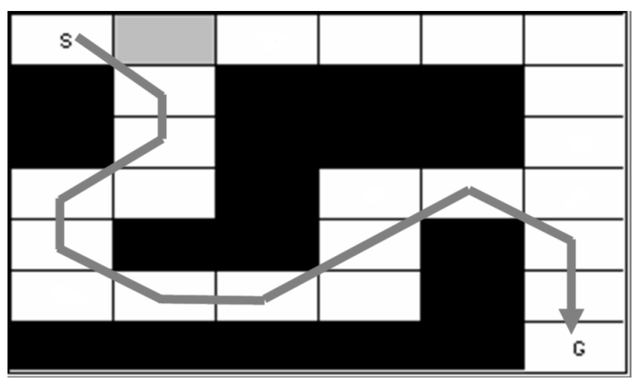
- **ARA***

ε =2.5



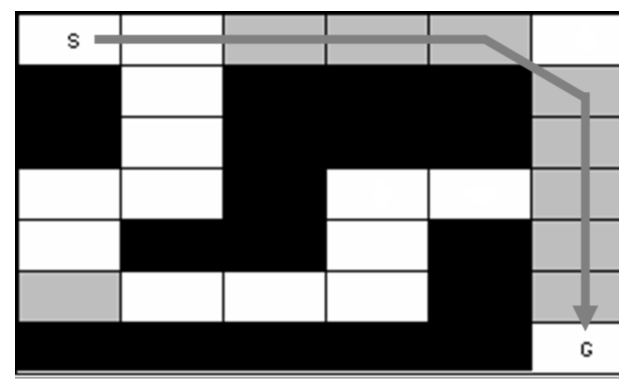
13 expansions
solution=11 moves

ε =1.5



1 expansion
solution=11 moves

ε =1.0



9 expansions
solution=10 moves

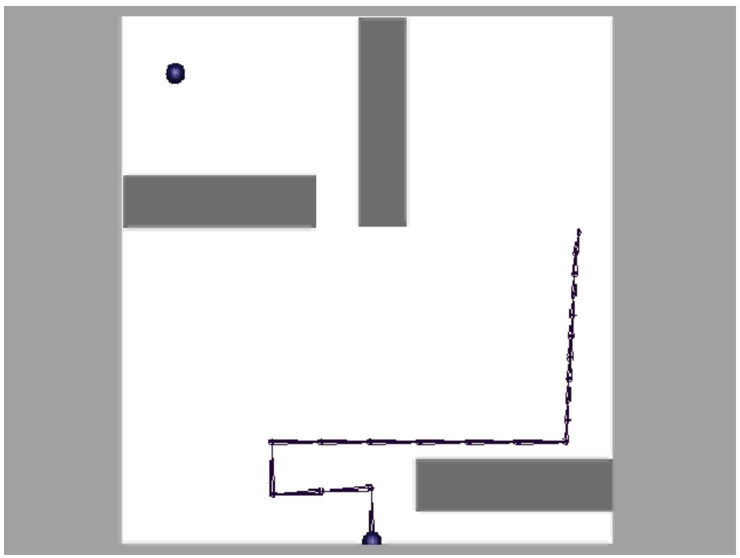
Maxim

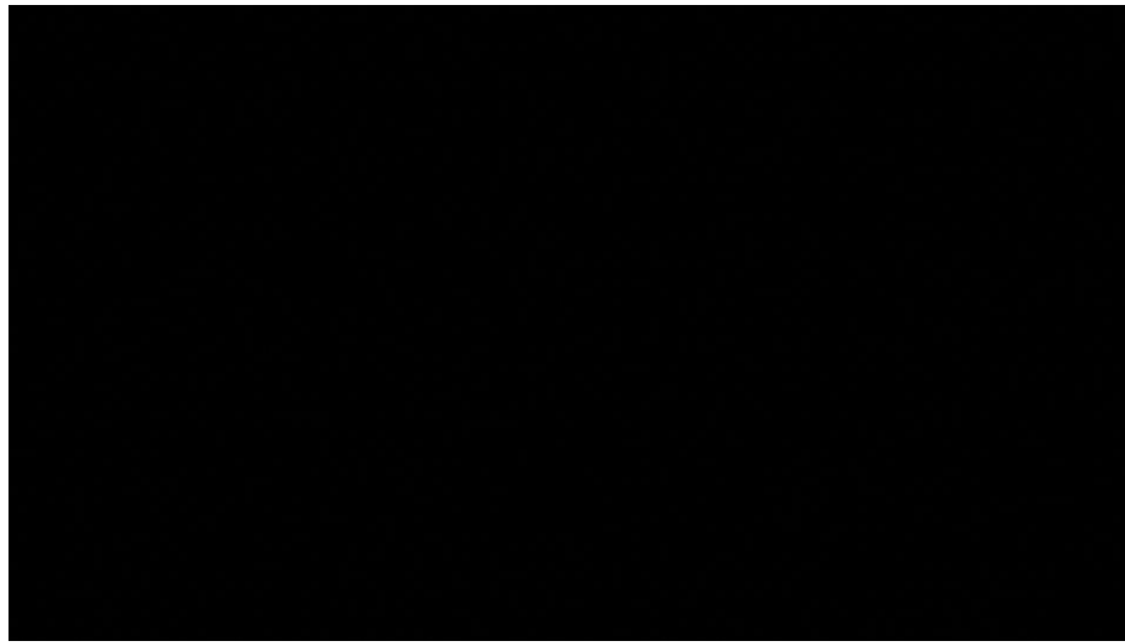
ARA*

- Motion planning for manipulators using ARA*:

Planning for 7DOF real robot arm

Planning for 20DOF planar arm





joint work with Willow Garage

Available online as part of ROS packages (SBPL arm planner)
ARA*/Anytime D* available as part of SBPL library

ARA*

- Planning for door opening using ARA*:

joint work with Willow Garage

Incremental Heuristic Search

- Fringe Saving A* (FSA*)
- Adaptive A* (AA*)
- Lifelong Planning A* (LPA*), D* Lite and Minimax LPA*
- Comparison of D* Lite and Adaptive A*
- Eager and Lazy Moving-Target Adaptive A* (MTAA*)
- Anytime Replanning A* (ARA*)
- Anytime D*

Anytime and Incremental Planning

Maxim

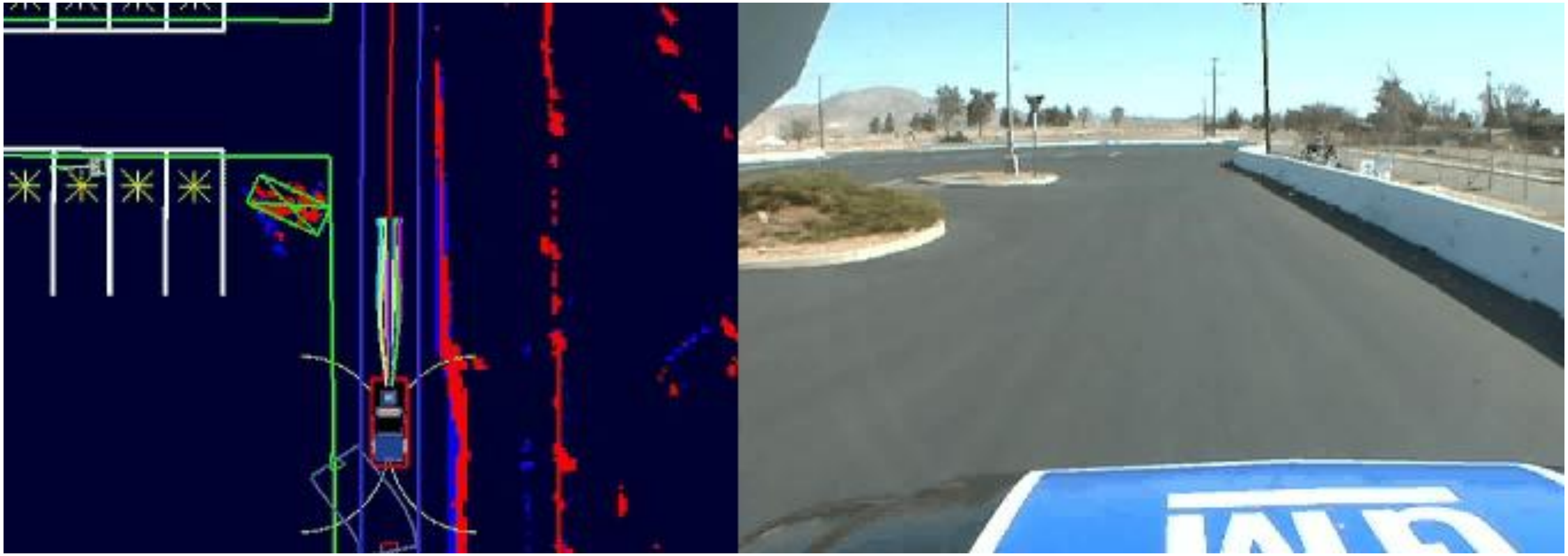
- Anytime D* [Likhachev, AIJ'08]: combination of ARA* and D* Lite
  - decreases ε and updates edge costs at the same time
  - re-computes a path by reusing previous state-values *(using a modified version of A* that reuses state values)*

set ε to large value;
until goal is reached
    ComputePathwithReuse(); *//modified to fix underconsistent states*
    publish ε-suboptimal path;
    follow the path until map is updated with new sensor information;
    update the corresponding edge costs;
    set s_{start} to the current state of the agent;
    if significant changes were observed
        increase ε or replan from scratch;
    else
        decrease ε;

Anytime and Incremental Planning

- 4D (x, y, Θ, V) planning using Anytime D* in Urban Challenge'07

Example of anytime planning



part of efforts by Tartanracing team from CMU for the Urban Challenge 2007 race

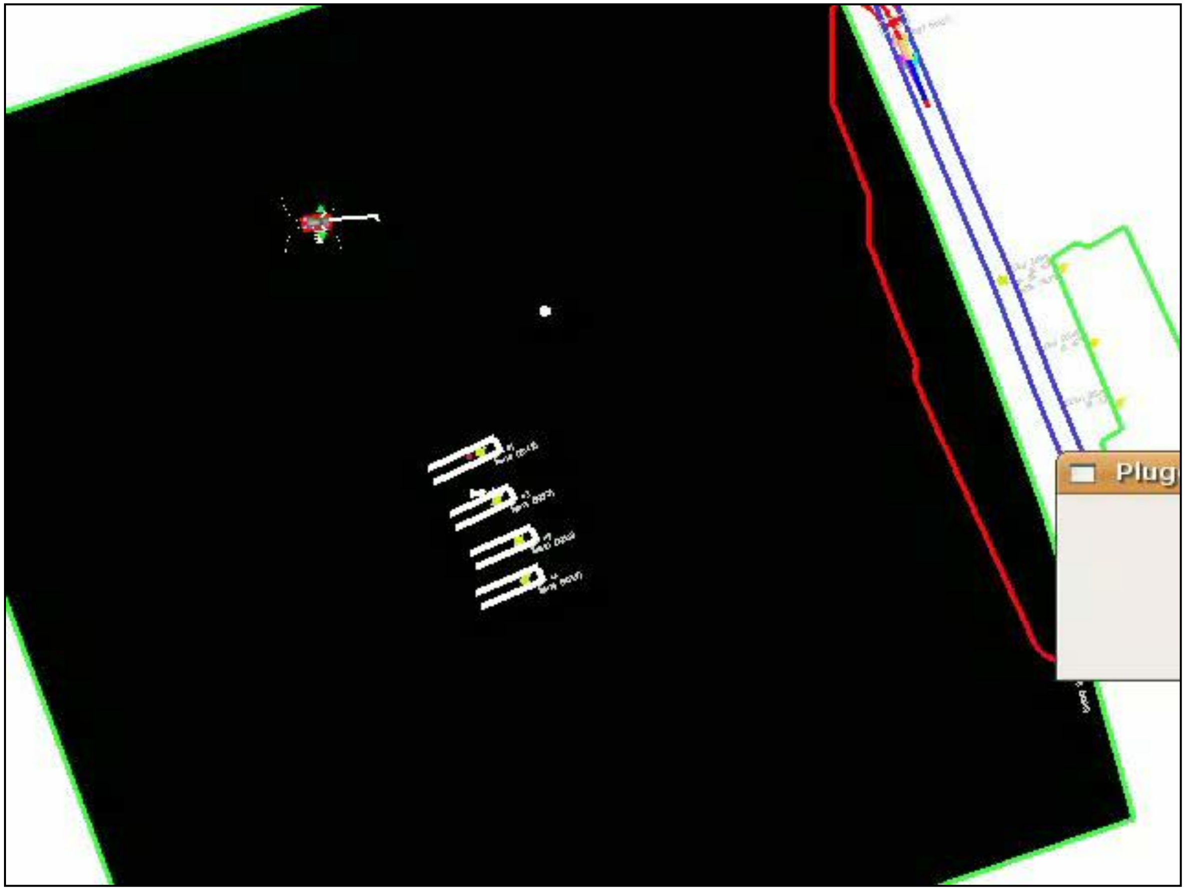
ARA*/Anytime D* and navigation planners using it are available as part of SBPL library
(as part of ROS packages and at www.seas.upenn.edu/~maximl/software.html)

Anytime and Incremental Planning

Maxim

- 4D (x, y, Θ, V) planning using Anytime D* in Urban Challenge'07

Example of re-planning



part of efforts by Tartanracing team from CMU for the Urban Challenge 2007 race

Table of Contents

Table of Contents

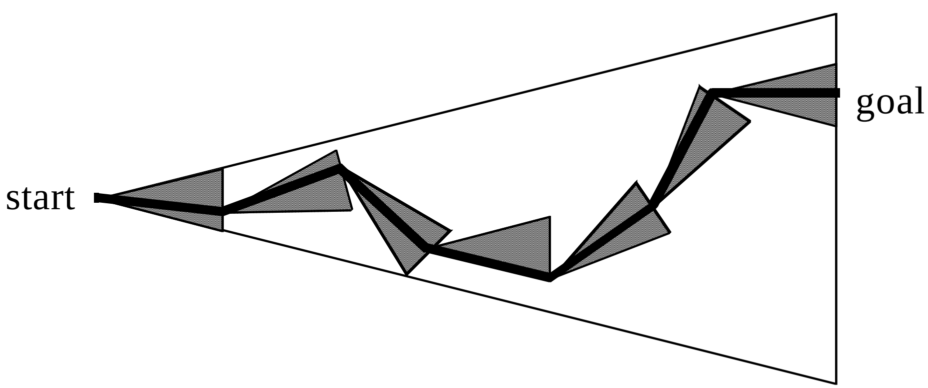
- Modeling Planning Domains
  - Graphs, MDPs
- Planning Problems and Strategies
  - Localization, Mapping, Navigation in Unknown Terrain
  - Agent-Centered Search, Assumptive Planning
- Efficient Implementations of Planning Strategies
  - Incremental Heuristic Search

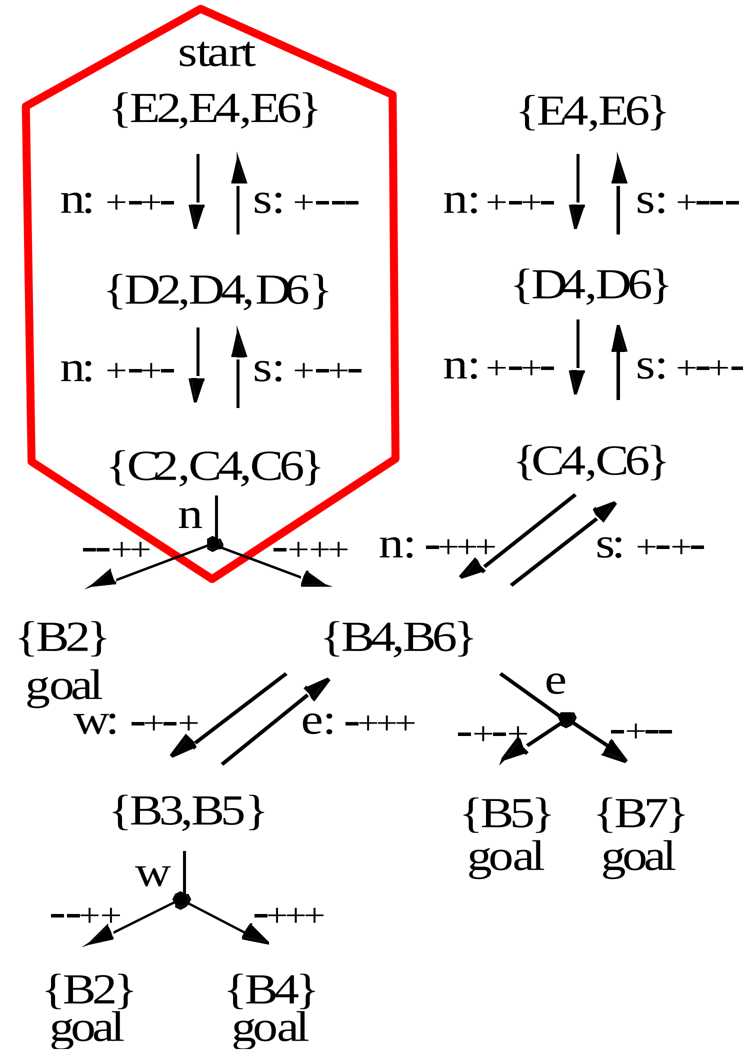
15 Minute Break

  - Real-Time Heuristic Search
  - Planning with Preferences on Uncertainty
  - Planning with Varying Abstractions

Greedy ~~Approx Optimal~~ Localization

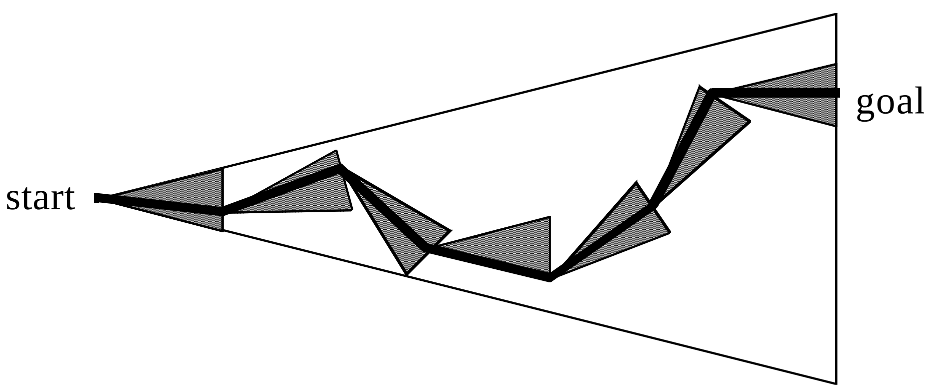
- Agent-centered search interleaves deterministic searches that result in a gain in information with action executions.

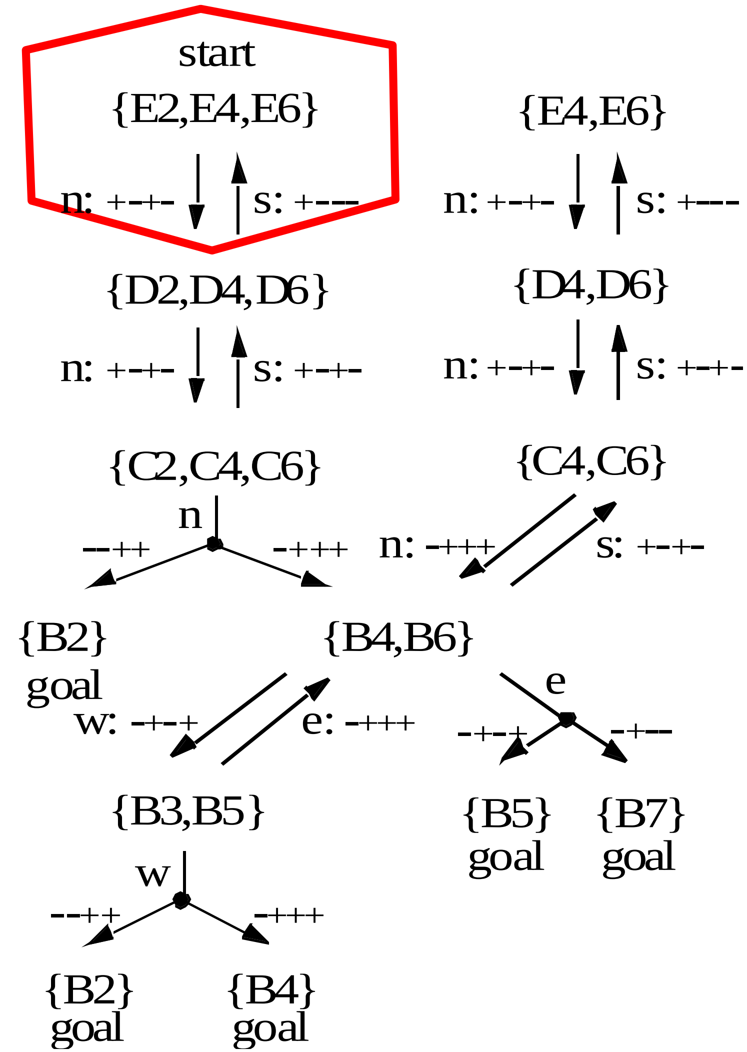




Real-Time Versions of A*

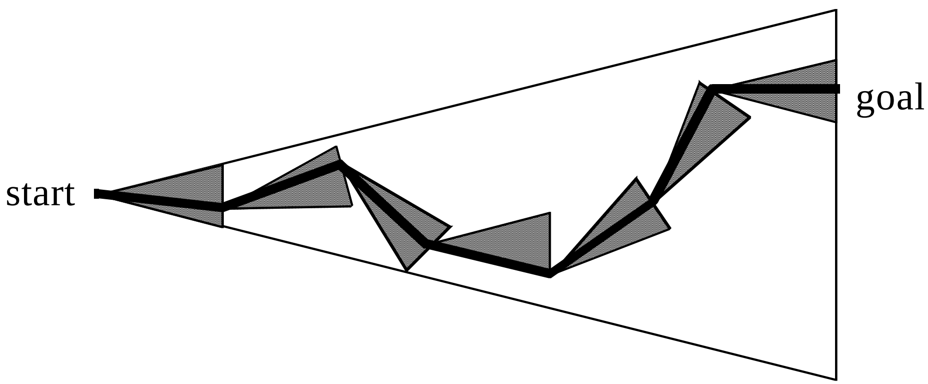
- Real-time search interleaves deterministic searches ~~that result in a gain in information~~ with action executions.

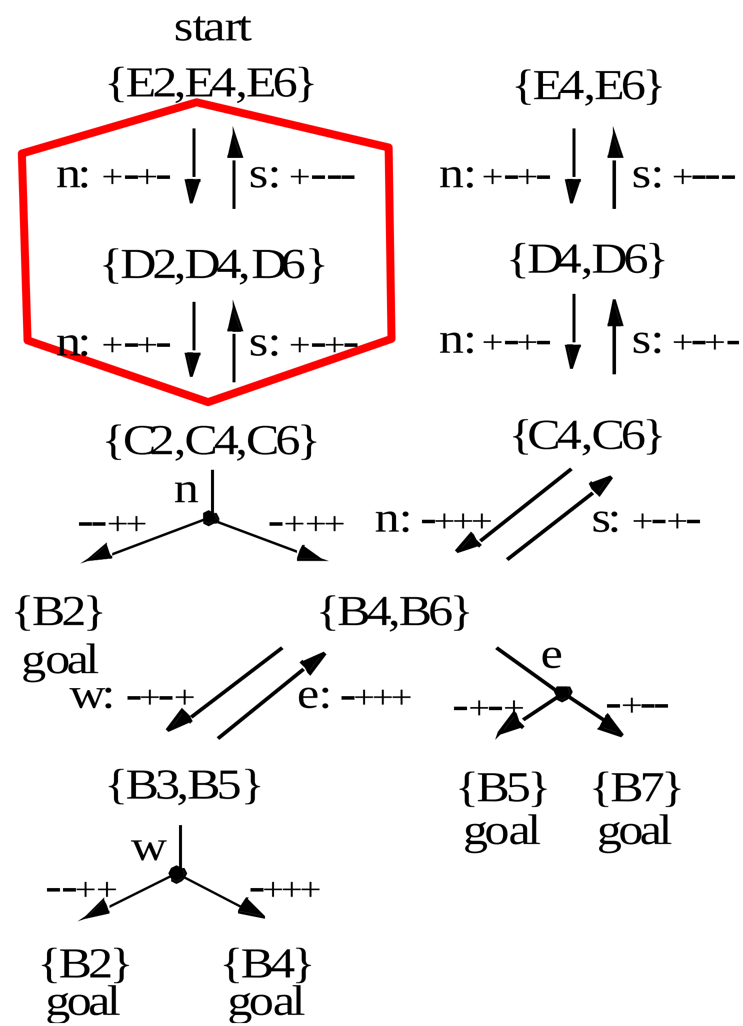




Real-Time Versions of A*

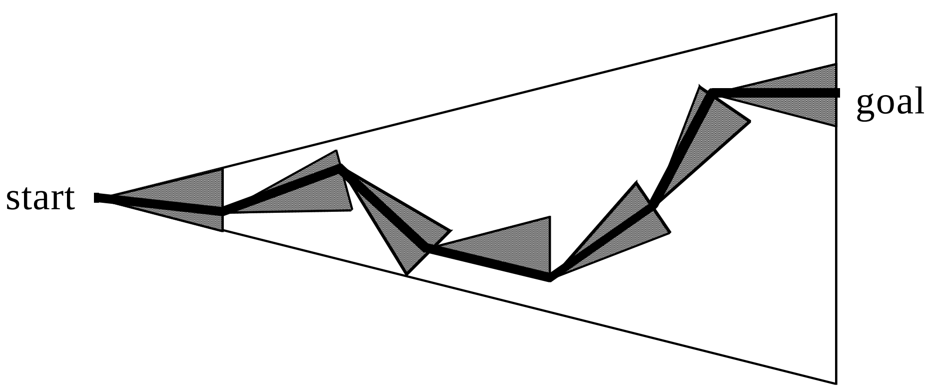
- Real-time search interleaves deterministic searches ~~that result in a gain in information~~ with action executions.

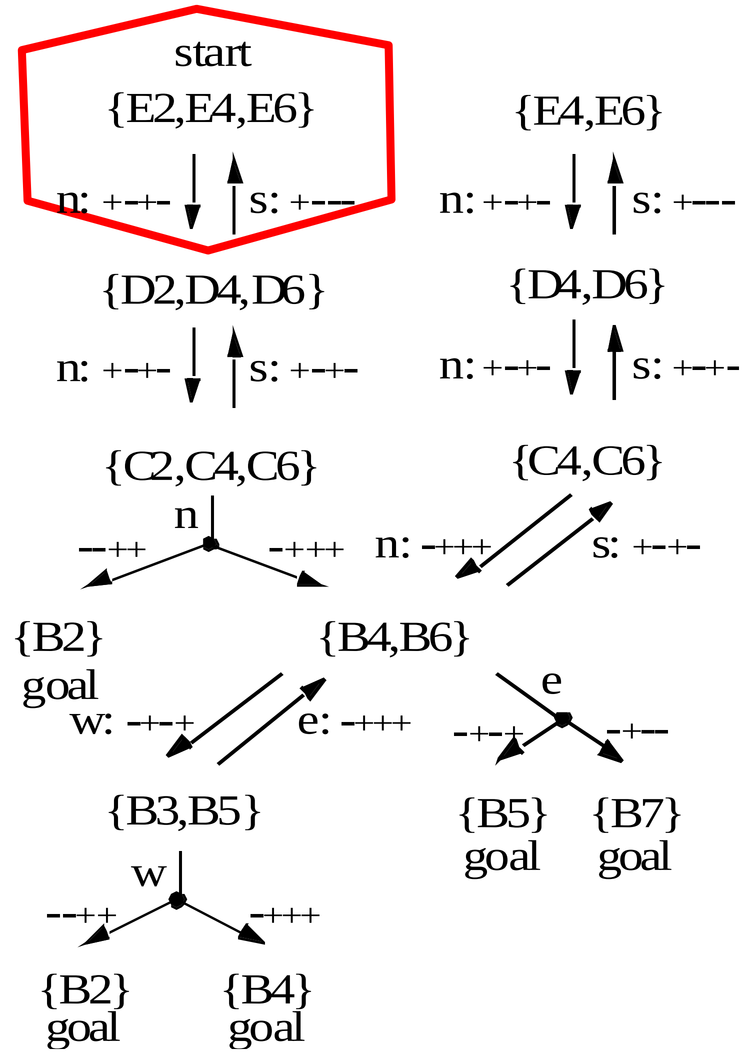




Real-Time Versions of A*

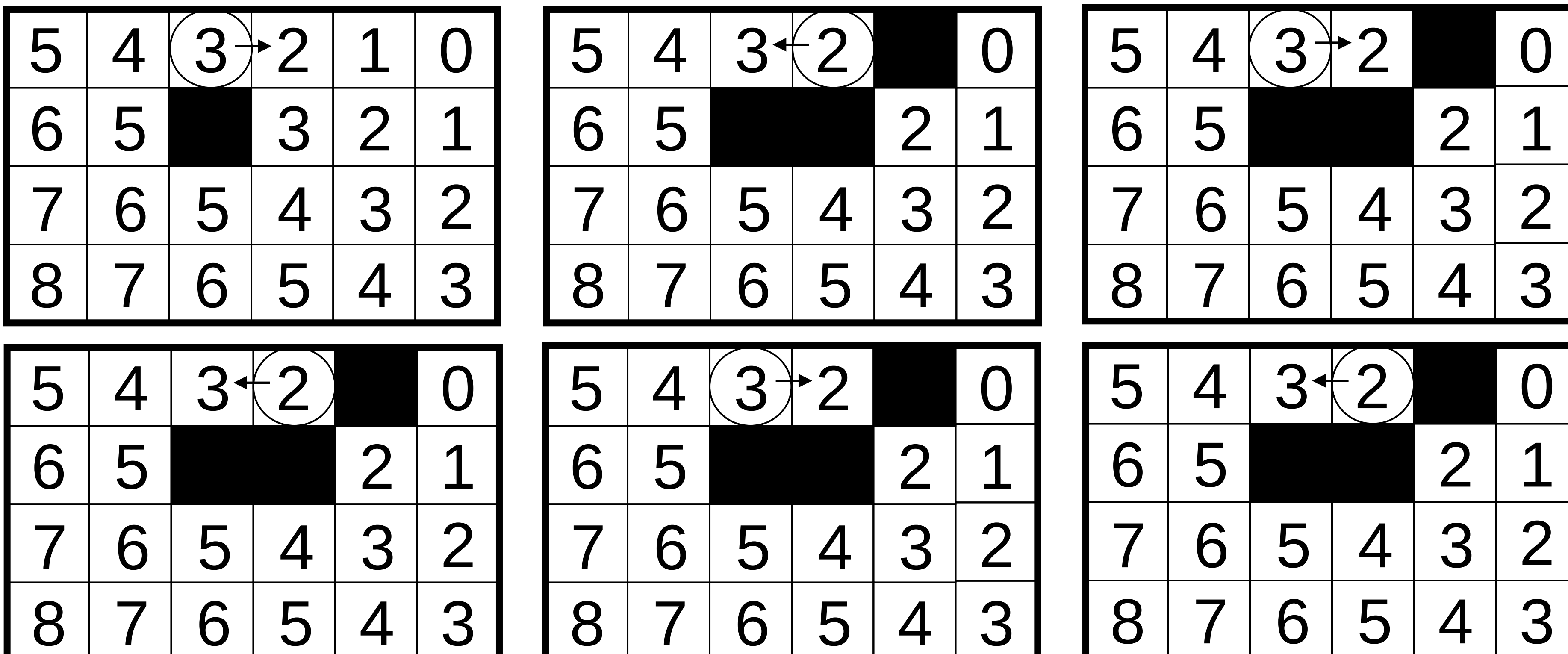
- Real-time search interleaves deterministic searches ~~that result in a gain in information~~ with action executions.





Real-Time Versions of A*

- One could repeatedly move to the most promising neighboring state, using the h-values.



4-neighbor grid

local minima are a problem

Real-Time Versions of A*

- Real-time heuristic search [Korf, 1990] solves search problems with a constant planning time between movements by interleaving partial searches around the robot cells with movements. It updates the h-values after every search to avoid cycling without reaching the goal. It typically does not follow a shortest path.
- There are many different real-time heuristic search algorithms. We present one of them.

Real-Time Heuristic Search

- Learning-Real Time A* (LRTA*)
- Comparison of D* Lite and LRTA*
- Real-Time Adaptive A* (RTAA*)
- Generalizations of LRTA*: Minimax LRTA* and RTDP

Learning Real-Time A* (LRTA*)

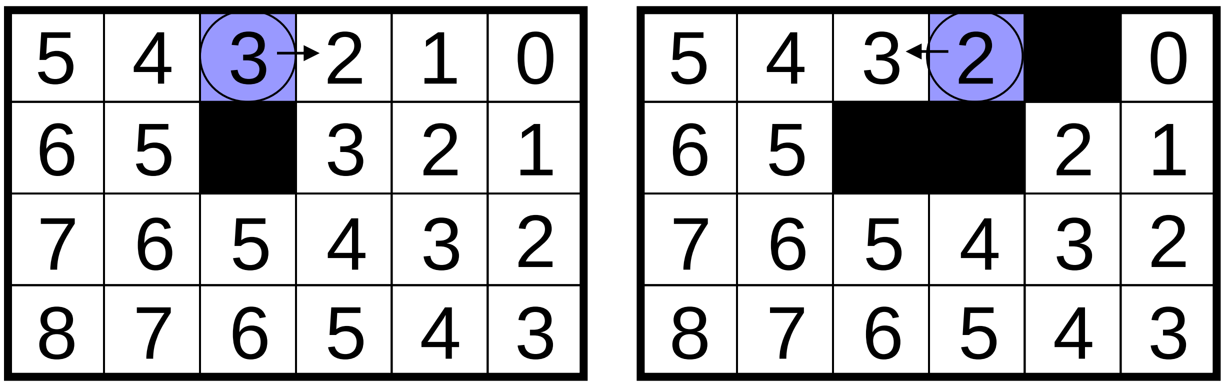
- LRTA* repeatedly moves to the most promising neighboring state, using and updating the h-values.

5	4	3	2	1	0
6	5		3	2	1
7	6	5	4	3	2
8	7	6	5	4	3

4-neighbor grid

Learning Real-Time A* (LRTA*)

- LRTA* repeatedly moves to the most promising neighboring state, using and updating the h-values.



4-neighbor grid

Learning Real-Time A* (LRTA*)

- LRTA* repeatedly moves to the most promising neighboring state, using and updating the h-values.

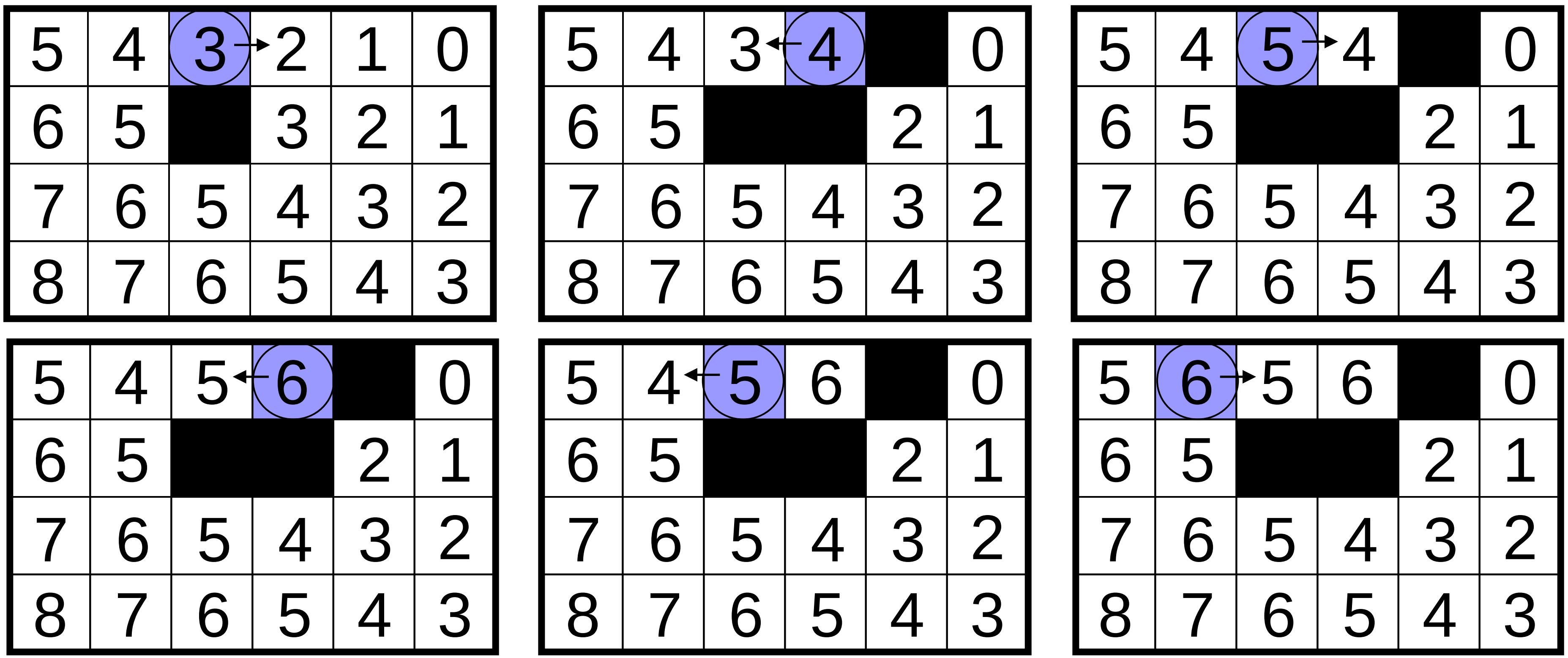
5	4	3	2	1	0
6	5		3	2	1
7	6	5	4	3	2
8	7	6	5	4	3

5	4	3	4		0
6	5			2	1
7	6	5	4	3	2
8	7	6	5	4	3

4-neighbor grid

Learning Real-Time A* (LRTA*)

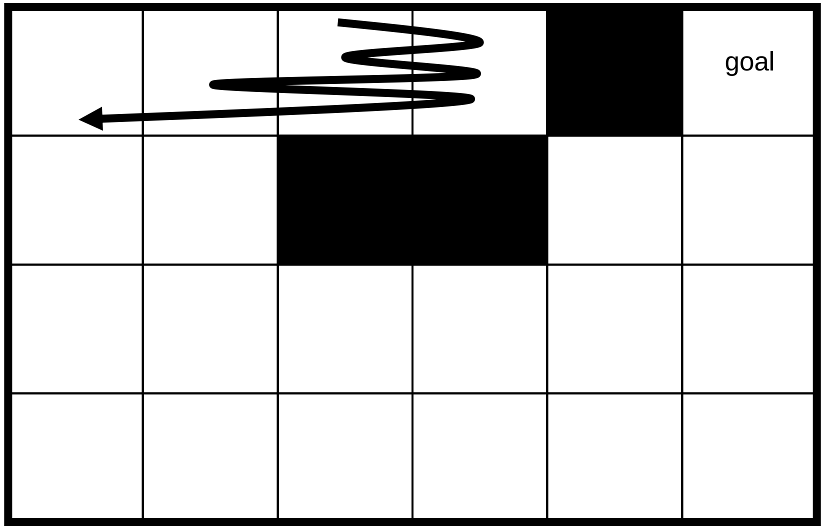
- LRTA* repeatedly moves to the most promising neighboring state, using and updating the h-values.



local minima are overcome by updating the h-values

Learning Real-Time A* (LRTA*)

- LRTA* repeatedly moves to the most promising neighboring state, using and updating the h-values.



4-neighbor grid

Learning Real-Time A* (LRTA*)

Properties of Learning Real-Time A* (LRTA*) [Korf, 1990]:

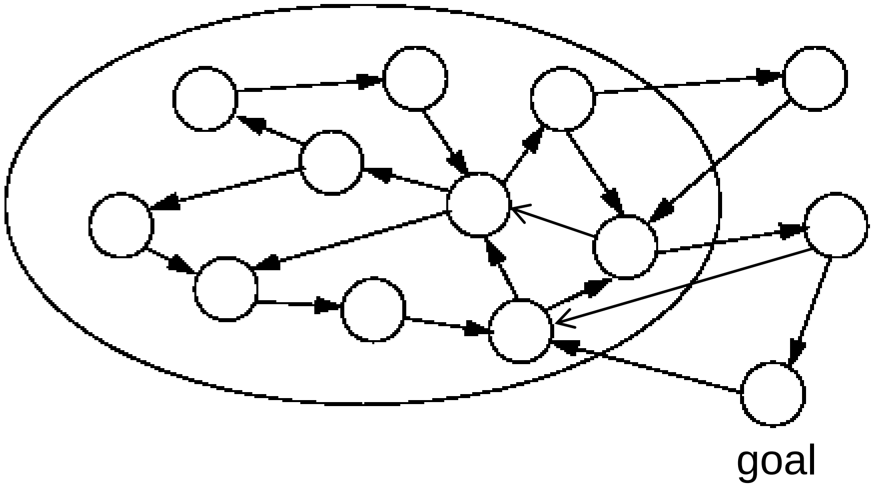
- The h-values of the same state are monotonically nondecreasing over time and thus indeed become more informed over time.
- The h-values remain consistent.
- The robot reaches the goal with $O(|V|^2)$ movements in safely explorable state spaces, where $|V|$ is the number of states (= unblocked cells) [Koenig, 2001].
- If the robot is reset into the start whenever it reaches the goal then the number of times that it does not follow a shortest path from the start to the goal is bounded from above by a constant if the cost increases are bounded from below by a positive constant.

Learning Real-Time A* (LRTA*)

- Theorem

  LRTA* reaches the goal if it is reachable from every state (= the search space is safely explorable).

- Proof:

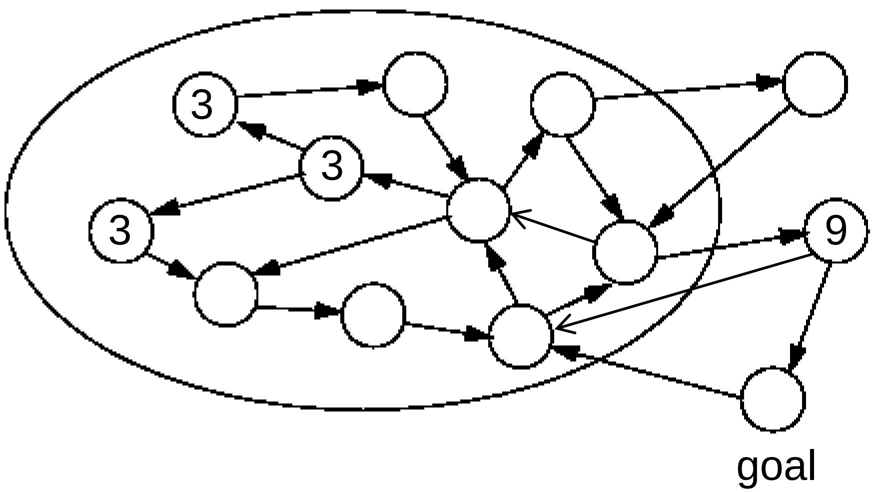


Learning Real-Time A* (LRTA*)

- Theorem

  LRTA* reaches the goal if it is reachable from every state (= the search space is safely explorable).

- Proof:

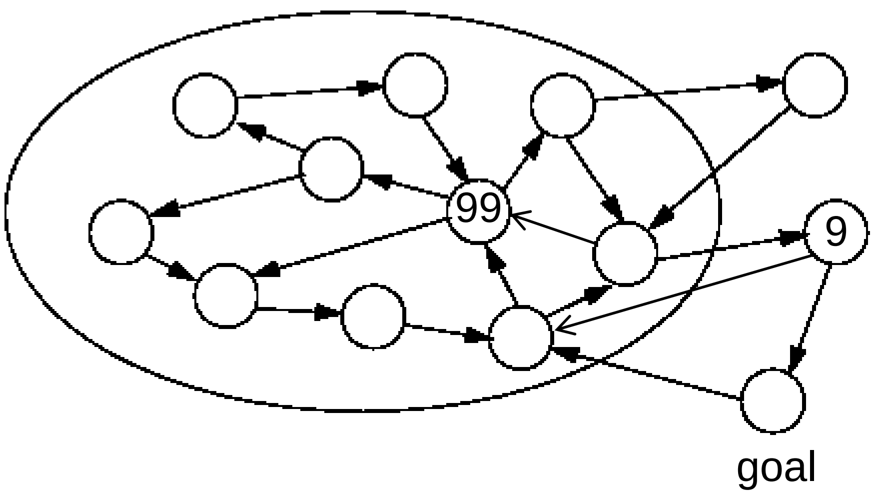


Learning Real-Time A* (LRTA*)

- Theorem

  LRTA* reaches the goal if it is reachable from every state (= the search space is safely explorable).

- Proof:



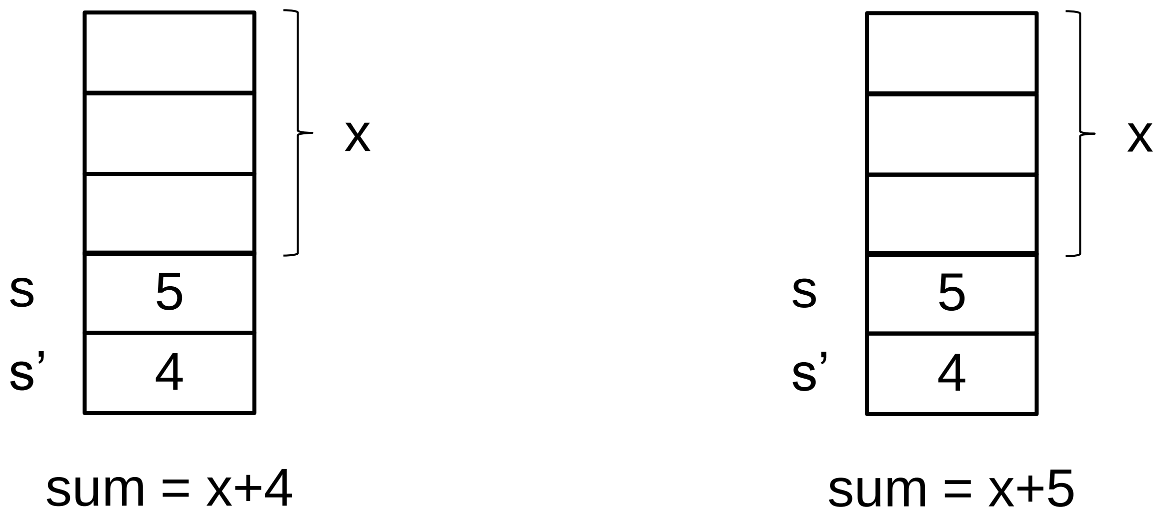
Learning Real-Time A* (LRTA*)

- Theorem [Koenig, 2001]

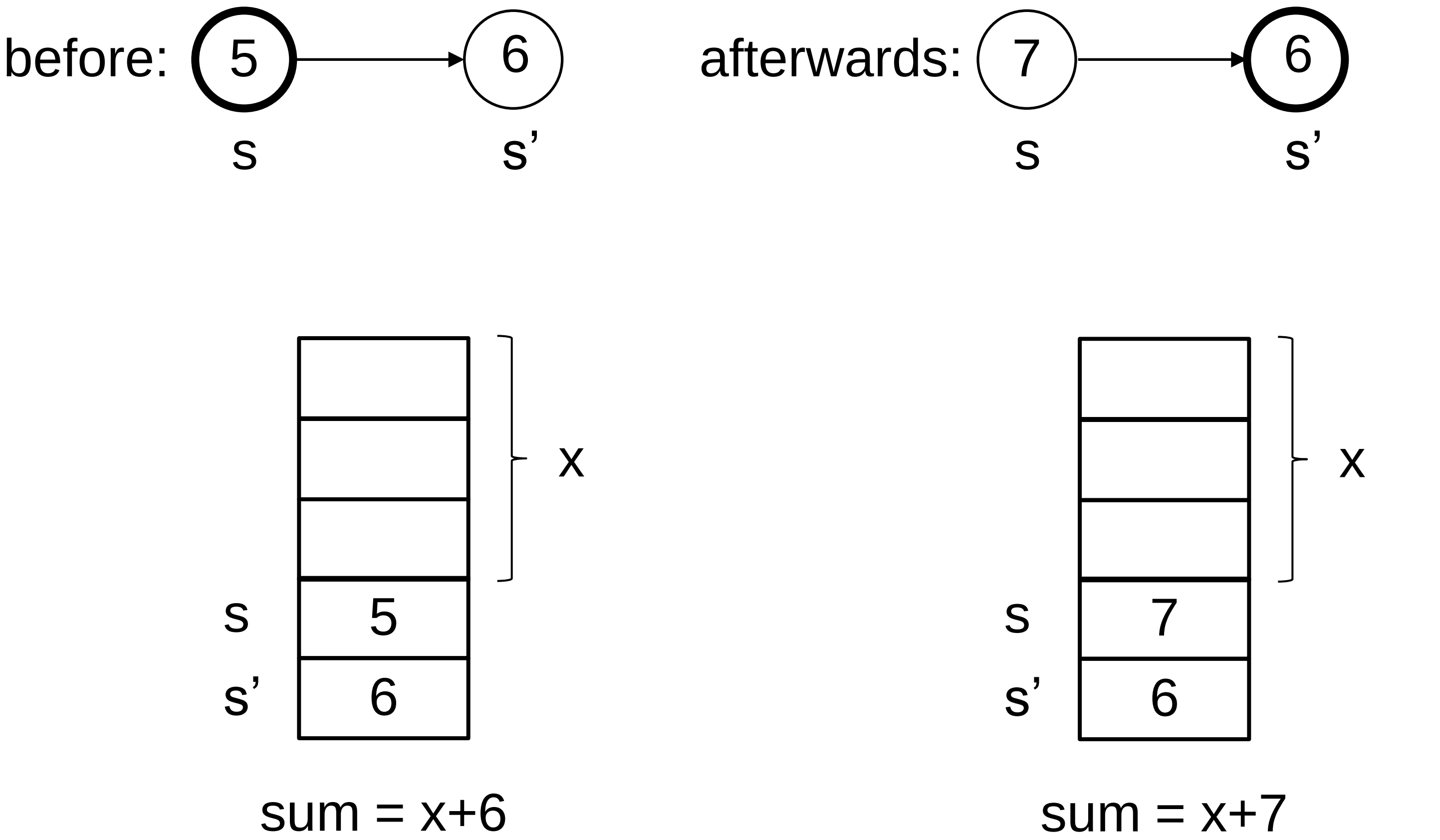
  The worst-case number of movements is O|V|^2) if the goal is reachable from every state and all movement costs are one, where |V| is the number of states (= unblocked cells).

- Proof under the assumption that all movements change the state: Consider the sum of all h-values minus the h-value of the robot state. The initial sum is at least zero. The final sum is at most |V| diameter since the h-value of every state is at most its goal distance. Every movement increases the sum by at least one.

Learning Real-Time A* (LRTA*)

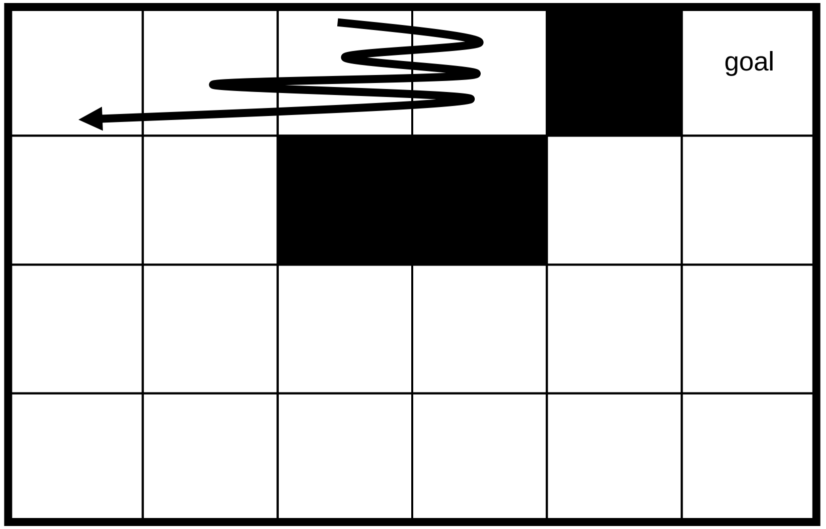


Learning Real-Time A* (LRTA*)



Learning Real-Time A* (LRTA*)

- LRTA* repeatedly moves to the most promising neighboring state, using and updating the h-values.



4-neighbor grid

Learning Real-Time A* (LRTA*)

We need larger lookaheads.
The possible design choices differ as follows:

- Which states to search?
- The h-values of which states to update?
- How many moves to make before the next search?

Learning Real-Time A* (LRTA*)

We need larger lookaheads.

We make the following design choices [Koenig, 2004]:

- Which states to search?
  The number x of states to search is determined by the available planning time between movements and is thus a parameter. We use the first x states expanded by an A* search. An A* search uses h-values to focus the search and always tries to disprove the path currently believed to be shortest.
- The h-values of which states to update?
  We use Dijkstra's algorithm to update the h-values of all x states searched.
- How many moves to make before the next search?
  We move the robot until it reaches a state different from the x states searched.

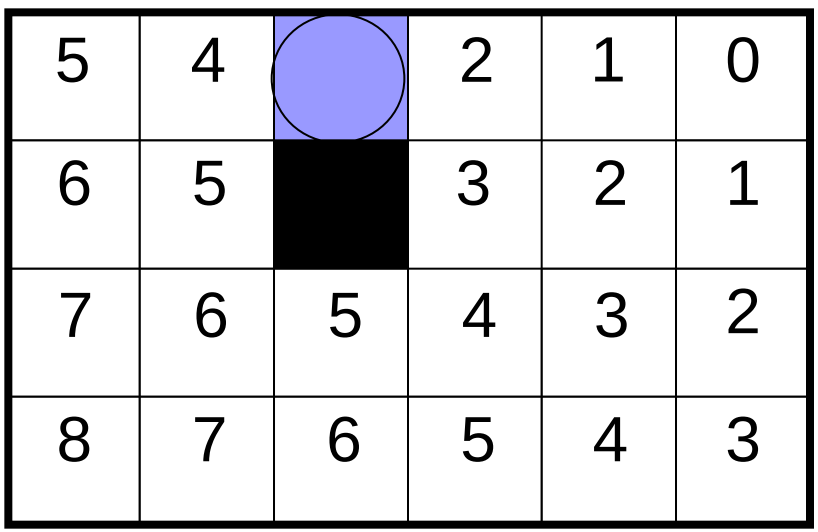
Learning Real-Time A* (LRTA*)

5	4	3	2	1	0
6	5		3	2	1
7	6	5	4	3	2
8	7	6	5	4	3

4-neighbor grid

Learning Real-Time A* (LRTA*)

- Step 1: Forward A* search

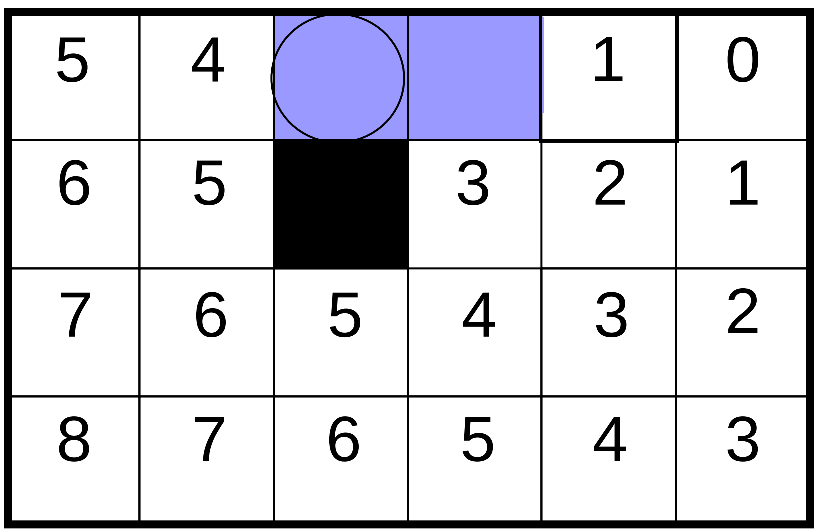


first A* state expansion

4-neighbor grid

Learning Real-Time A* (LRTA*)

- Step 1: Forward A* search

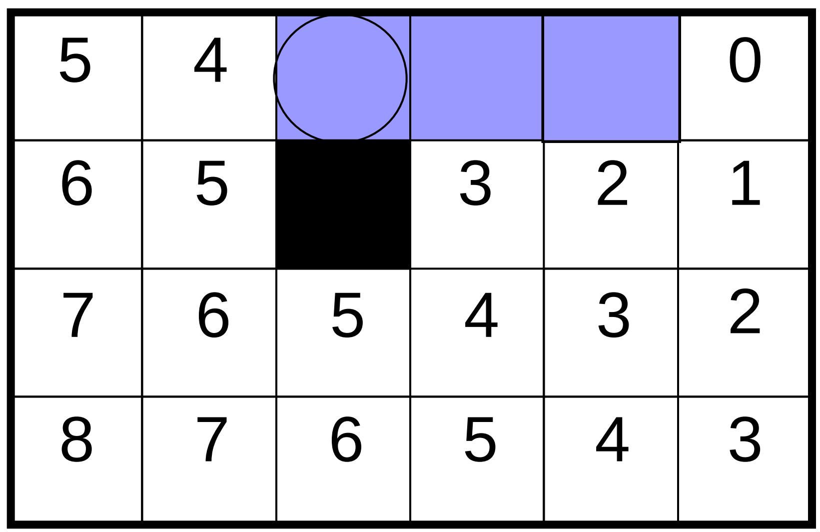


second A* state expansion

4-neighbor grid

Learning Real-Time A* (LRTA*)

- Step 1: Forward A* search

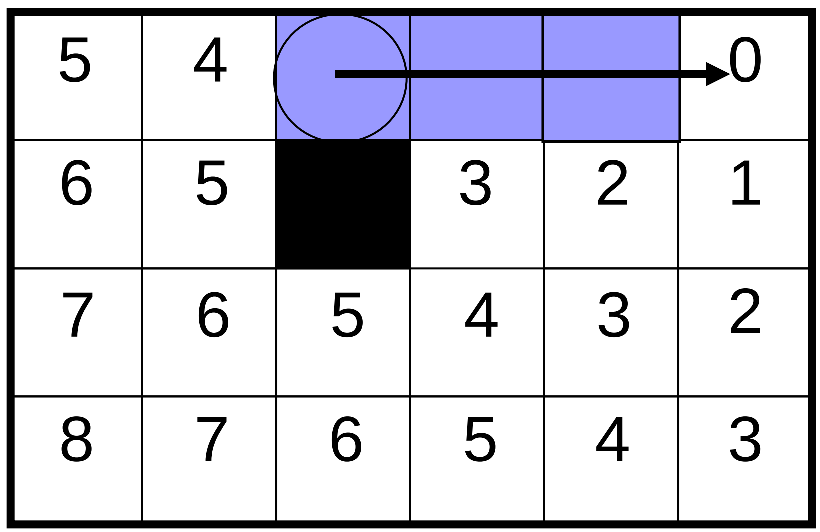


third A* state expansion

4-neighbor grid

Learning Real-Time A* (LRTA*)

- Step 1: Forward A* search

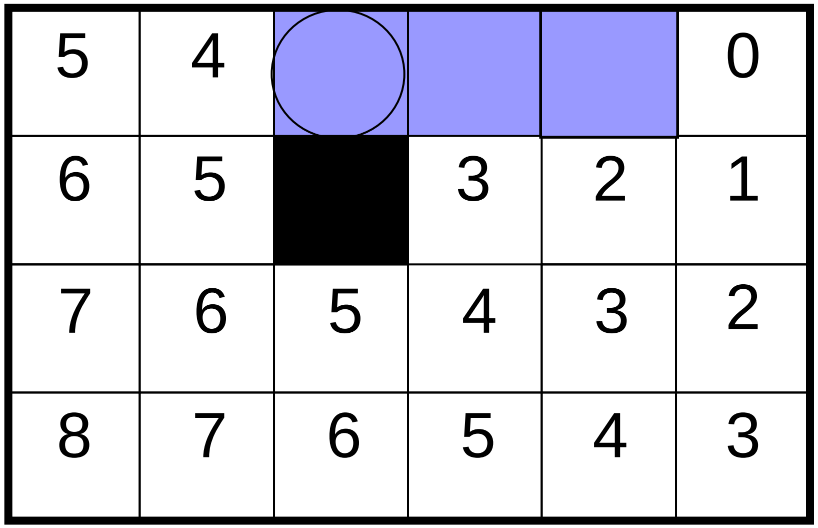


third A* state expansion

4-neighbor grid

Learning Real-Time A* (LRTA*)

- Step 1: Forward A* search



third A* state expansion

Learning Real-Time A* (LRTA*)

- Step 2: Updating the h-values with Dijkstra's algorithm

5	4	∞	∞	∞	0
6	5		3	2	1
7	6	5	4	3	2
8	7	6	5	4	3

first iteration of Dijkstra's algorithm

4-neighbor grid

Learning Real-Time A* (LRTA*)

- Step 2: Updating the h-values with Dijkstra's algorithm

5	4	∞	∞	1	0
6	5		3	2	1
7	6	5	4	3	2
8	7	6	5	4	3

second iteration of Dijkstra's algorithm

4-neighbor grid

Learning Real-Time A* (LRTA*)

- Step 2: Updating the h-values with Dijkstra's algorithm

5	4	∞	2	1	0
6	5		3	2	1
7	6	5	4	3	2
8	7	6	5	4	3

third iteration of Dijkstra's algorithm

4-neighbor grid

Learning Real-Time A* (LRTA*)

- Step 2: Updating the h-values with Dijkstra's algorithm

5	4	∞	2	1	0
6	5		3	2	1
7	6	5	4	3	2
8	7	6	5	4	3

fourth iteration of Dijkstra's algorithm

4-neighbor grid

Learning Real-Time A* (LRTA*)

- Step 2: Updating the h-values with Dijkstra's algorithm

5	4	3	2	1	0
6	5		3	2	1
7	6	5	4	3	2
8	7	6	5	4	3

fifth iteration of Dijkstra's algorithm

4-neighbor grid

Learning Real-Time A* (LRTA*)

- Step 2: Updating the h-values with Dijkstra's algorithm

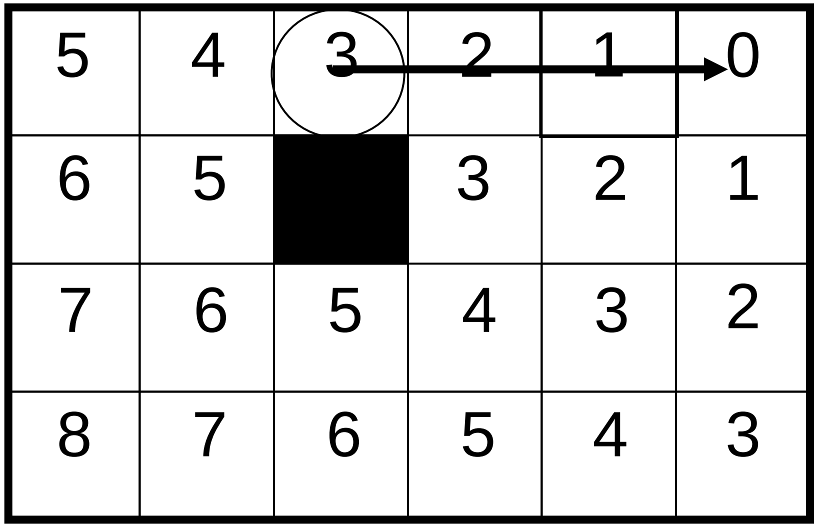
5	4	3	2	1	0
6	5		3	2	1
7	6	5	4	3	2
8	7	6	5	4	3

sixth iteration of Dijkstra's algorithm

4-neighbor grid

Learning Real-Time A* (LRTA*)

- Step 3: Moving along the path

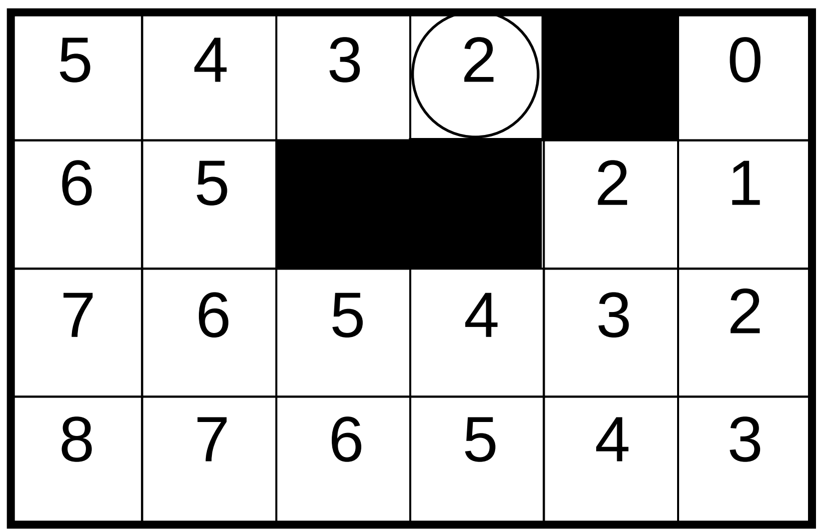


follow the path

4-neighbor grid

Learning Real-Time A* (LRTA*)

- Step 3: Moving along the path

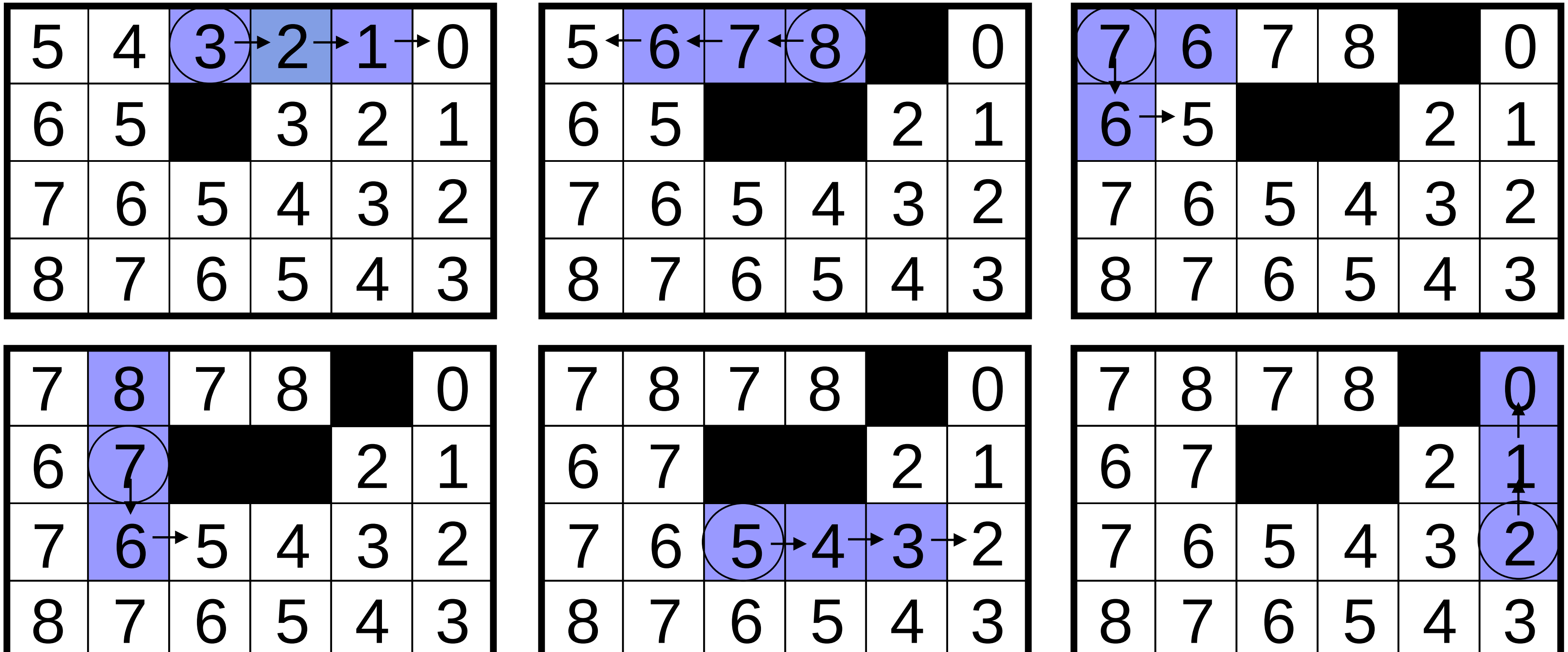


follow the path

4-neighbor grid

Learning Real-Time A* (LRTA*)

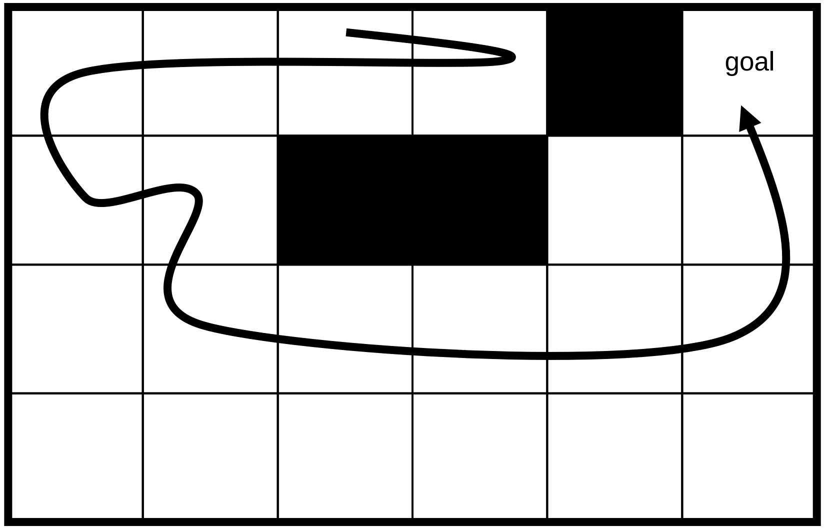
- LRTA* repeatedly moves to the most promising neighboring state, using and updating the h-values with a lookahead > 1.



4-neighbor grid

Learning Real-Time A* (LRTA*)

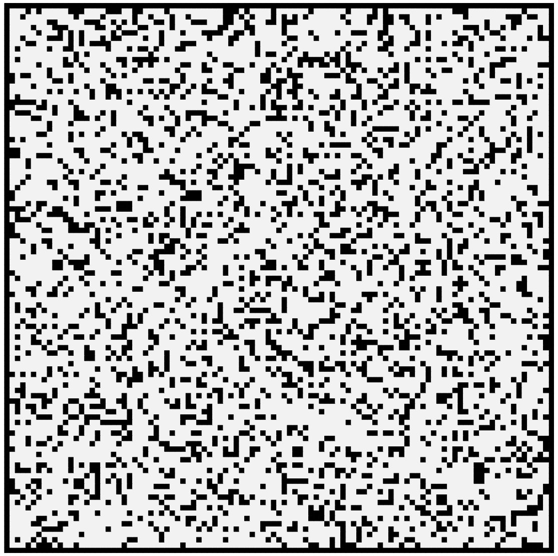
- LRTA* repeatedly moves to the most promising neighboring state, using and updating the h-values with a lookahead > 1.



4-neighbor grid

Learning Real-Time A* (LRTA*)

- Safely explorable random grids of size 301 x 301



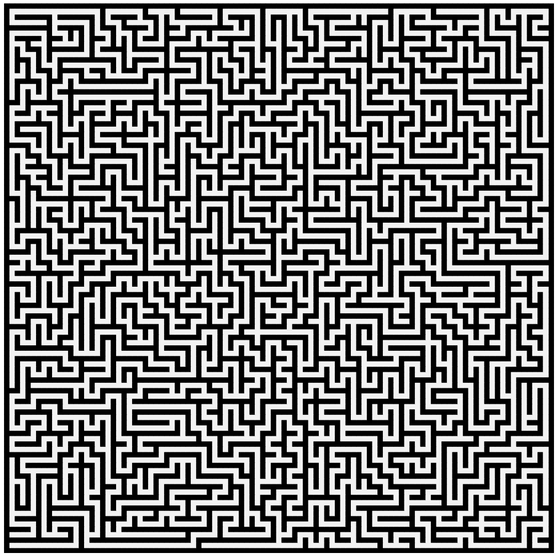
Grids with 25% random obstacles
The h-values are generally not misleading.
Larger lookaheads are less helpful.

Learning Real-Time A* (LRTA*)

lookahead	Manhattan distance		octile distance	
	planning time	move-ments	planning time	move-ments
1	28280	499	28293	363
11	28698	315	28878	315
21	29153	302	29477	311
31	29615	299	…	…
41	…	…	…	…

Learning Real-Time A* (LRTA*)

- DFS mazes of size 301 x 301



Acyclic mazes generated with DFS
The h-values are generally misleading.
Larger lookaheads are very helpful.

Learning Real-Time A* (LRTA*)

lookahead	Manhattan distance		octile distance	
	planning time	move-ments	planning time	move-ments
1	985362	1987574	628175	1259958
11	313998	337704	277974	272842
21	279856	205370	273280	177143
31	…	…	310131	135554
41	…	…	348330	114917

Real-Time Heuristic Search

- Learning-Real Time A* (LRTA*)
- Comparison of D* Lite and LRTA*
- Real-Time Adaptive A* (RTAA*)
- Generalizations of LRTA*: Minimax LRTA* and RTDP

LRTA* vs D* Lite

D* Lite

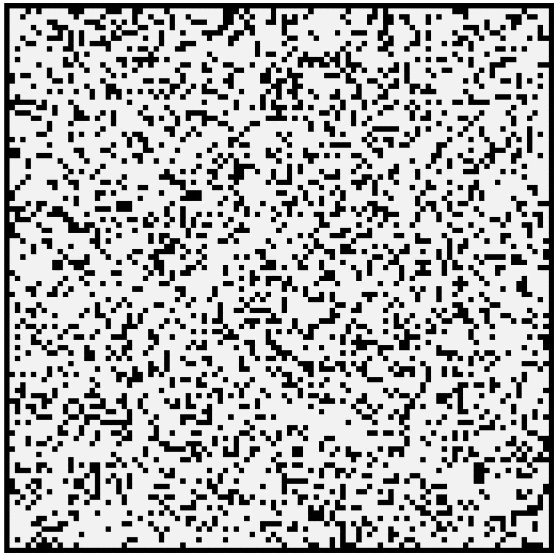
- can detect that the goal is unreachable,
- cannot satisfy hard real-time requirements and
- has a worst-case number of movements of $O(|V| \log |V|)$.

LRTA*

- cannot easily detect that the goal is unreachable,
- can satisfy hard real-time requirements and
- has a worst-case number of movements of $\theta(|V|^2)$.

LRTA* vs D* Lite

- Safely explorable random grids of size 301 x 301



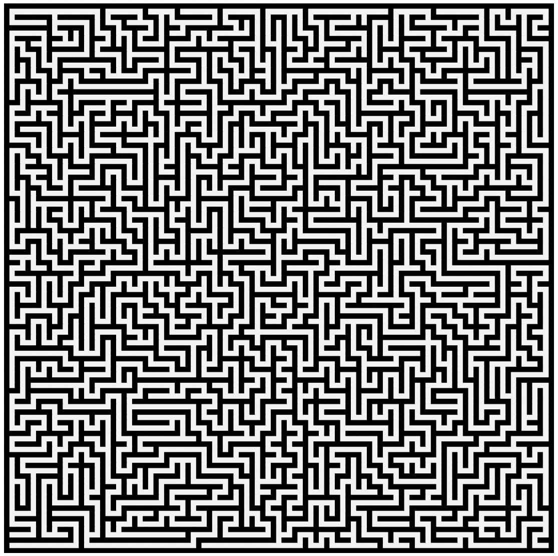
Grids with 25% random obstacles
The h-values are generally not misleading.
Larger lookaheads are less helpful.

LRTA* vs D* Lite

lookahead	Manhattan distance		octile distance	
	planning time	move-ments	planning time	move-ments
D* Lite	36826	309	40737	314
1	28280	499	28293	363
11	28698	315	28878	315
21	29153	302	29477	311
31	29615	299	…	…
41	…	…	…	…

LRTA* vs D* Lite

- DFS mazes of size 301 x 301



Acyclic mazes generated with DFS
The h-values are generally misleading.
Larger lookaheads are very helpful.

LRTA* vs D* Lite

lookahead	Manhattan distance		octile distance	
	planning time	move-ments	planning time	move-ments
D* Lite	357417	21738	373561	21140
1	985362	1987574	628175	1259958
11	313998	337704	277974	272842
21	279856	205370	273280	177143
31	…	…	310131	135554
41	…	…	348330	114917

Real-Time Heuristic Search

- Learning-Real Time A* (LRTA*)
- Comparison of D* Lite and LRTA*
- Real-Time Adaptive A* (RTAA*)
- Generalizations of LRTA*: Minimax LRTA* and RTDP

Real-Time Adaptive A* (RTAA*)

- We use AA* to create Real-Time Adaptive A* (RTAA*) [Koenig and Likhachev, 2006], a real-time heuristic search method with similar properties as LRTA*. RTAA* improves on LRTA* by updating the h-values much faster although they are not quite as informed.

Real-Time Adaptive A* (RTAA*)

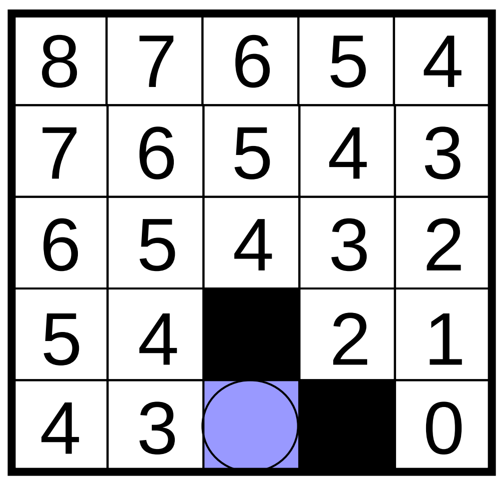
- LRTA* step 1: forward A* search

8	7	6	5	4
7	6	5	4	3
6	5	4	3	2
5	4		2	1
4	3	2		0

4-neighbor grid

Real-Time Adaptive A* (RTAA*)

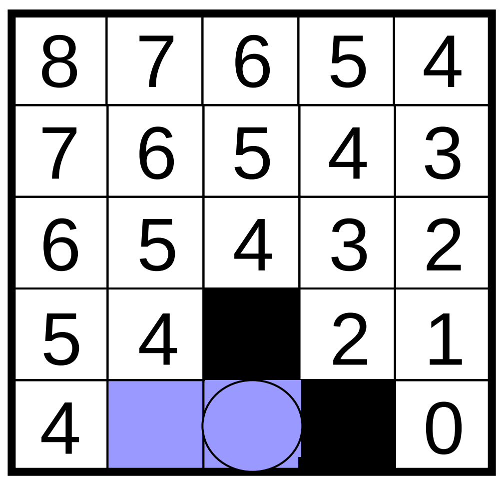
- LRTA* step 1: forward A* search



4-neighbor grid

Real-Time Adaptive A* (RTAA*)

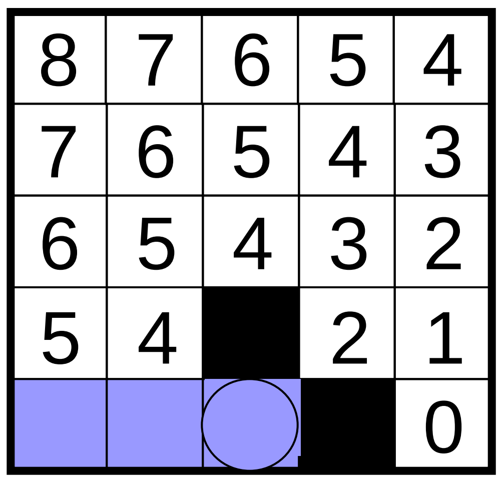
- LRTA* step 1: forward A* search



4-neighbor grid

Real-Time Adaptive A* (RTAA*)

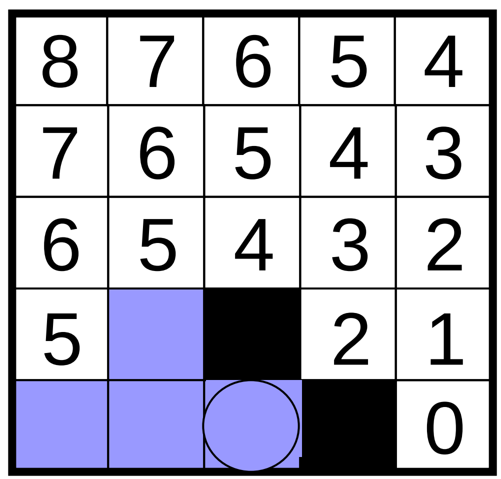
- LRTA* step 1: forward A* search



4-neighbor grid

Real-Time Adaptive A* (RTAA*)

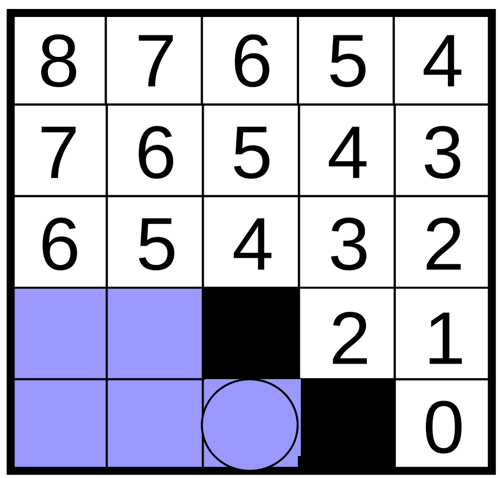
- LRTA* step 1: forward A* search



4-neighbor grid

Real-Time Adaptive A* (RTAA*)

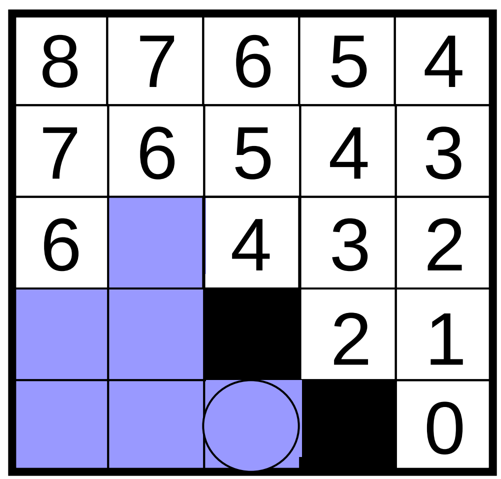
- LRTA* step 1: forward A* search



4-neighbor grid

Real-Time Adaptive A* (RTAA*)

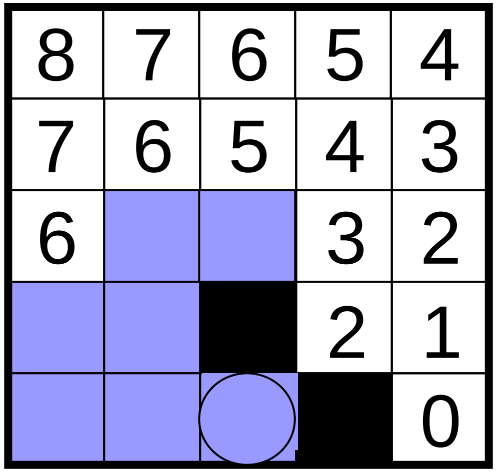
- LRTA* step 1: forward A* search



4-neighbor grid

Real-Time Adaptive A* (RTAA*)

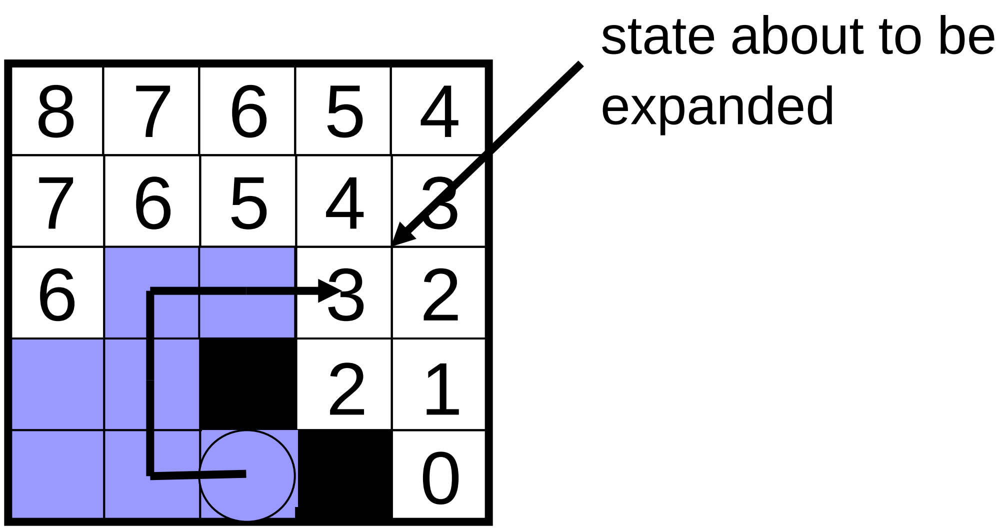
- LRTA* step 1: forward A* search



4-neighbor grid

Real-Time Adaptive A* (RTAA*)

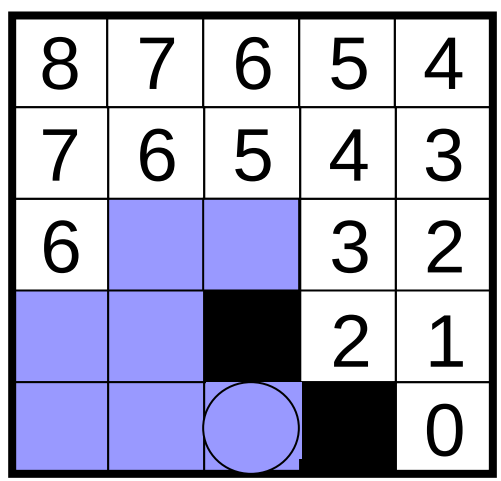
- LRTA* step 1: forward A* search



4-neighbor grid

Real-Time Adaptive A* (RTAA*)

- LRTA* step 2: updating the h-values



4-neighbor grid

Real-Time Adaptive A* (RTAA*)

- LRTA* step 2: updating the h-values

8	7	6	5	4
7	6	5	4	3
6	∞	∞	3	2
∞	∞		2	1
∞	∞	∞		0

4-neighbor grid

Real-Time Adaptive A* (RTAA*)

- LRTA* step 2: updating the h-values

8	7	6	5	4
7	6	5	4	3
6	∞	4	3	2
∞	∞		2	1
∞	∞	∞		0

4-neighbor grid

Real-Time Adaptive A* (RTAA*)

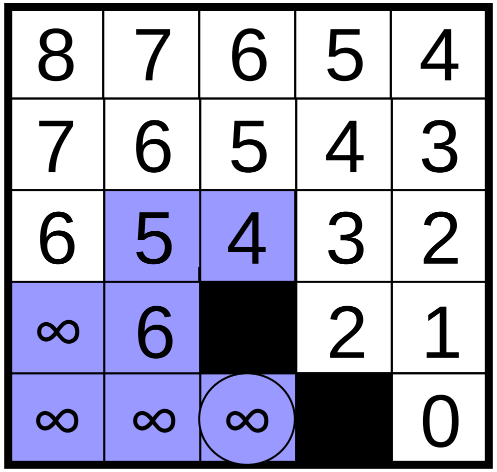
- LRTA* step 2: updating the h-values

8	7	6	5	4
7	6	5	4	3
6	5	4	3	2
∞	∞		2	1
∞	∞	∞		0

4-neighbor grid

Real-Time Adaptive A* (RTAA*)

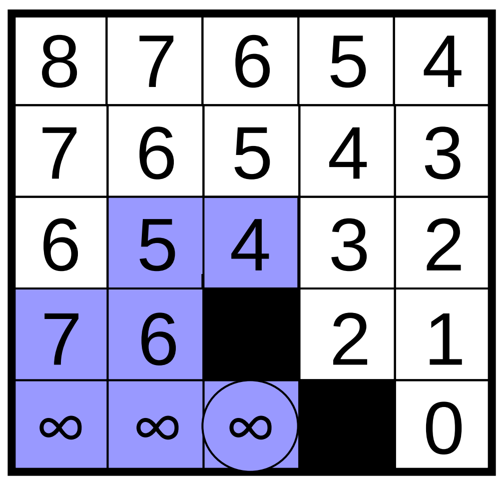
- LRTA* step 2: updating the h-values



4-neighbor grid

Real-Time Adaptive A* (RTAA*)

- LRTA* step 2: updating the h-values



4-neighbor grid

Real-Time Adaptive A* (RTAA*)

- LRTA* step 2: updating the h-values

8	7	6	5	4
7	6	5	4	3
6	5	4	3	2
7	6		2	1
∞	7	∞		0

4-neighbor grid

Real-Time Adaptive A* (RTAA*)

- LRTA* step 2: updating the h-values

8	7	6	5	4
7	6	5	4	3
6	5	4	3	2
7	6		2	1
8	7	∞		0

4-neighbor grid

Real-Time Adaptive A* (RTAA*)

- LRTA* step 2: updating the h-values

8	7	6	5	4
7	6	5	4	3
6	5	4	3	2
7	6		2	1
8	7	8		0

4-neighbor grid

Real-Time Adaptive A* (RTAA*)

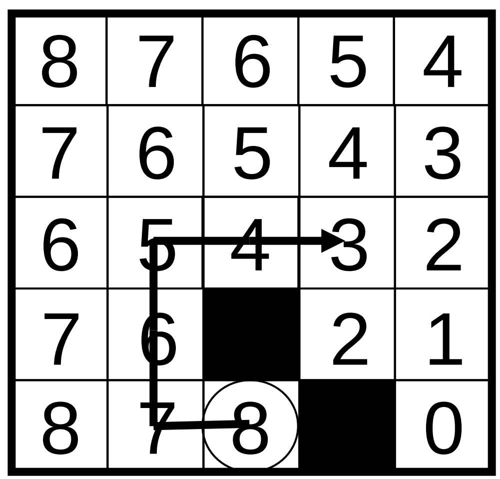
- LRTA* step 2: updating the h-values

8	7	6	5	4
7	6	5	4	3
6	5	4	3	2
7	6		2	1
8	7	8		0

4-neighbor grid

Real-Time Adaptive A* (RTAA*)

- LRTA* step 3: moving along the path



Real-Time Adaptive A* (RTAA*)

- LRTA* step 3: moving along the path

8	7	6	5	4
7	6	5	4	3
6	5	4	3	2
7	6		2	1
8	7	8		0

4-neighbor grid

Real-Time Adaptive A* (RTAA*)

- LRTA* step 3: moving along the path

8	7	6	5	4
7	6	5	4	3
6	5	4	3	2
7	6		2	1
8	7	8		0

4-neighbor grid

Real-Time Adaptive A* (RTAA*)

- LRTA* step 3: moving along the path

8	7	6	5	4
7	6	5	4	3
6	5		3	2
7	6		2	1
8	7	8		0

4-neighbor grid

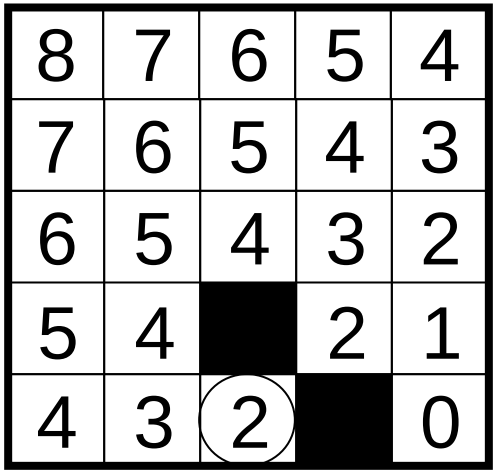
Real-Time Adaptive A* (RTAA*)

Properties of LRTA* [Korf, 1990]

- The h-values of the same state are monotonically nondecreasing over time and thus indeed become more informed over time.
- The h-values remain consistent.
- The robot reaches the goal in safely explorable state spaces.
- If the robot is reset into the start whenever it reaches the goal then the number of times that it does not follow a shortest path from the start to the goal is bounded from above by a constant if the cost increases are bounded from below by a positive constant.

Real-Time Adaptive A* (RTAA*)

- RTAA* step 1: forward A* search



4-neighbor grid

Real-Time Adaptive A* (RTAA*)

- RTAA* step 1: forward A* search

8	7	6	5	4
7	6	5	4	3
6	5	4	3	2
5	4		2	1
4	3	**0**		0

bold = g-value
regular = h-value

4-neighbor grid

Real-Time Adaptive A* (RTAA*)

- RTAA* step 1: forward A* search

8	7	6	5	4
7	6	5	4	3
6	5	4	3	2
5	4		2	1
4	**1**	**0**		0

bold = g-value

regular = h-value

4-neighbor grid

Real-Time Adaptive A* (RTAA*)

- RTAA* step 1: forward A* search

8	7	6	5	4
7	6	5	4	3
6	5	4	3	2
5	4		2	1
2	**1**	**0**		0

bold = g-value

regular = h-value

4-neighbor grid

Real-Time Adaptive A* (RTAA*)

- RTAA* step 1: forward A* search

8	7	6	5	4
7	6	5	4	3
6	5	4	3	2
5	**2**		2	1
2	**1**	**0**		0

bold = g-value
regular = h-value

4-neighbor grid

Real-Time Adaptive A* (RTAA*)

- RTAA* step 1: forward A* search

8	7	6	5	4
7	6	5	4	3
6	5	4	3	2
3	**2**		2	1
2	**1**	**0**		0

bold = g-value

regular = h-value

4-neighbor grid

Real-Time Adaptive A* (RTAA*)

- RTAA* step 1: forward A* search

8	7	6	5	4
7	6	5	4	3
6	**3**	4	3	2
3	**2**		2	1
2	**1**	**0**		0

bold = g-value

regular = h-value

4-neighbor grid

Real-Time Adaptive A* (RTAA*)

- RTAA* step 1: forward A* search

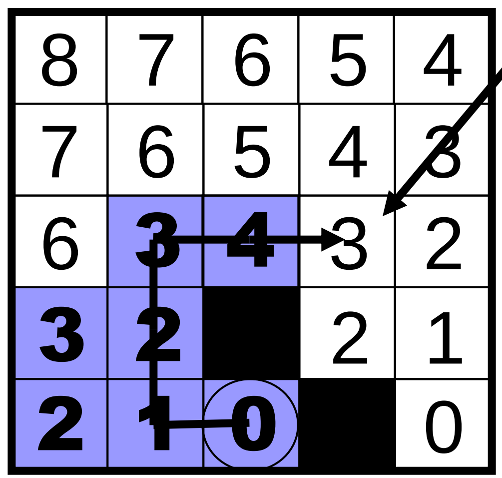
8	7	6	5	4
7	6	5	4	3
6	**3**	**4**	3	2
3	**2**		2	1
2	**1**	**0**		0

bold = g-value
regular = h-value

4-neighbor grid

Real-Time Adaptive A* (RTAA*)

- RTAA* step 1: forward A* search



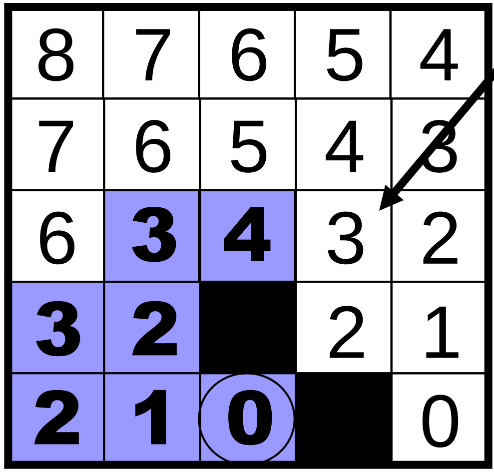
state about to be expanded
g-value = 5
h-value = 3
f-value = 8

bold = g-value
regular = h-value

4-neighbor grid

Real-Time Adaptive A* (RTAA*)

- RTAA* step 2: updating the h-values
  - RTAA*: For each expanded state s: set $h_{new}(s)$ = f(~~goal~~) – g(s). f(state about to be expanded)
  - LRTA*: For each expanded state s: use Dijkstra to determine $h_{new}(s)$.



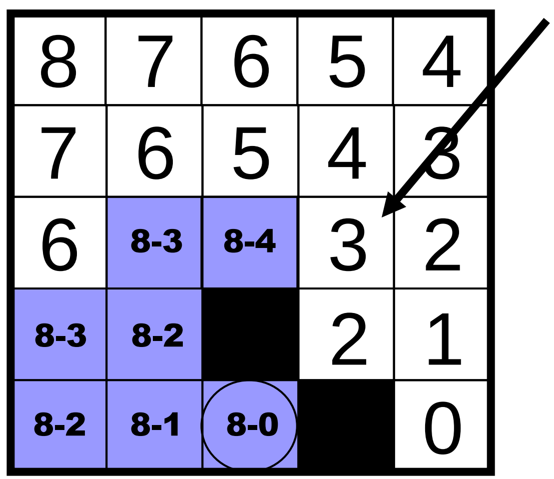
state about to be expanded
g-value = 5
h-value = 3
f-value = 8

bold = g-value
regular = h-value

4-neighbor grid

Real-Time Adaptive A* (RTAA*)

- RTAA* step 2: updating the h-values

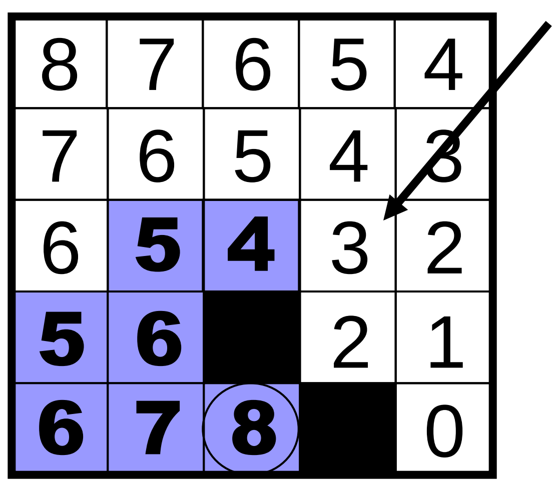


state about to be expanded
g-value = 5
h-value = 3
f-value = 8

4-neighbor grid

Real-Time Adaptive A* (RTAA*)

- RTAA* step 2: updating the h-values



state about to be expanded
g-value = 5
h-value = 3
f-value = 8

4-neighbor grid

Real-Time Adaptive A* (RTAA*)

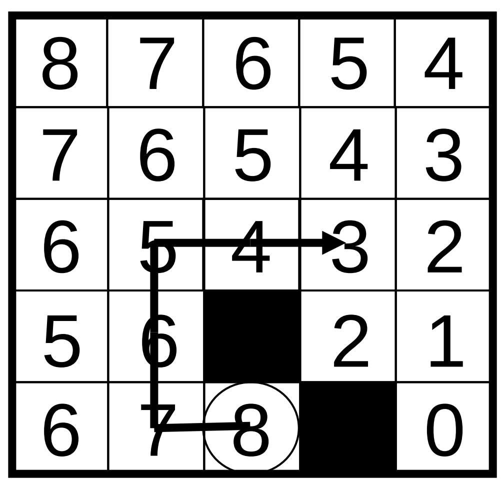
- RTAA* step 2: updating the h-values

8	7	6	5	4
7	6	5	4	3
6	5	4	3	2
5	6		2	1
6	7	8		0

4-neighbor grid

Real-Time Adaptive A* (RTAA*)

- RTAA* step 3: moving along the path



4-neighbor grid

Real-Time Adaptive A* (RTAA*)

- RTAA* step 3: moving along the path

8	7	6	5	4
7	6	5	4	3
6	5	4	3	2
5	6		2	1
6	7	8		0

4-neighbor grid

Real-Time Adaptive A* (RTAA*)

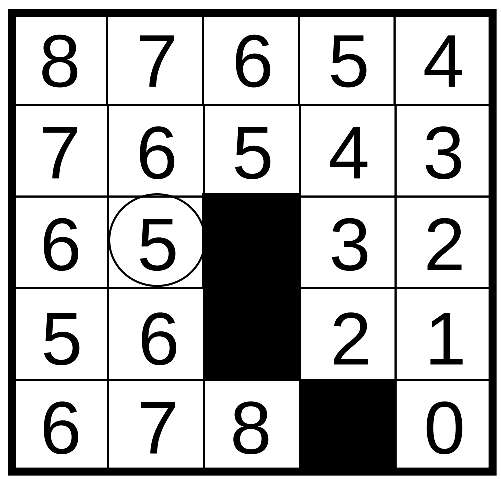
- RTAA* step 3: moving along the path

8	7	6	5	4
7	6	5	4	3
6	5	4	3	2
5	6		2	1
6	7	8		0

4-neighbor grid

Real-Time Adaptive A* (RTAA*)

- RTAA* step 3: moving along the path



4-neighbor grid

Real-Time Adaptive A* (RTAA*)

Properties of RTAA* [Koenig and Likhachev, 2006]

- The h-values of the same state are monotonically nondecreasing over time and thus indeed become more informed over time.
- The h-values remain consistent.
- The robot reaches the goal in safely explorable state spaces.
- If the robot is reset into the start whenever it reaches the goal then the number of times that it does not follow a shortest path from the start to the goal is bounded from above by a constant if the cost increases are bounded from below by a positive constant.

Real-Time Adaptive A* (RTAA*)

- RTAA*

8	7	6	5	4
7	6	5	4	3
6	5		3	2
5	6		2	1
6	7	8		0

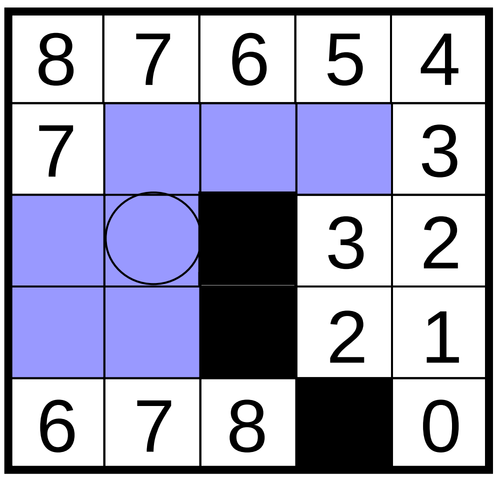
- LRTA*

8	7	6	5	4
7	6	5	4	3
6	5		3	2
7	6		2	1
8	7	8		0

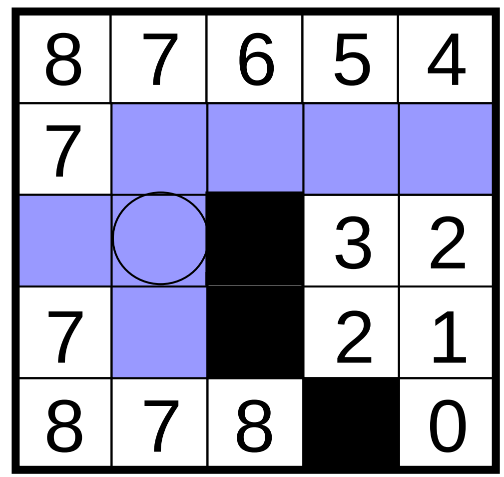
4-neighbor grid

Real-Time Adaptive A* (RTAA*)

- RTAA*



- LRTA*



4-neighbor grid

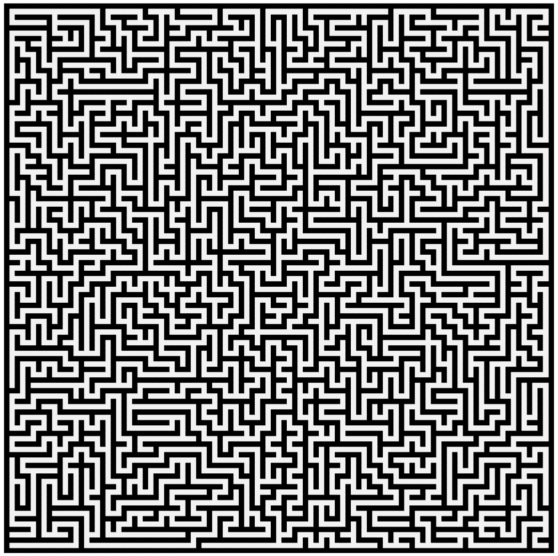
Real-Time Adaptive A* (RTAA*)

Relationship of RTAA* and LRTA*

- RTAA* with only one expanded state per A* search behaves exactly like LRTA* with only one expanded state per A* search.
- If RTAA* and LRTA* have the same h-values before they update the h-values then the h-values of RTAA* after the update are dominated by the h-values of LRTA*.

Real-Time Adaptive A* (RTAA*)

- DFS mazes of size 151 x 151



Real-Time Adaptive A* (RTAA*)

	RTAA*			LRTA*		
	expansions	move-ments	planning time per search [ms]	expansions	move-ments	planning time per search [ms]
1	248538	248538	0.20	248538	248538	0.27
9	104229	56708	2.01	87613	47291	2.80
17	85866	33853	4.37	79313	30470	6.25
25	89258	26338	6.86	82851	23270	10.23
33	96840	22022	9.41	92908	20016	14.31
41	105703	18629	11.99	102788	17274	18.50
49	117036	16638	14.46	113140	15398	22.67
57	128560	15367	16.83	125013	14285	26.69

+7%

-59%

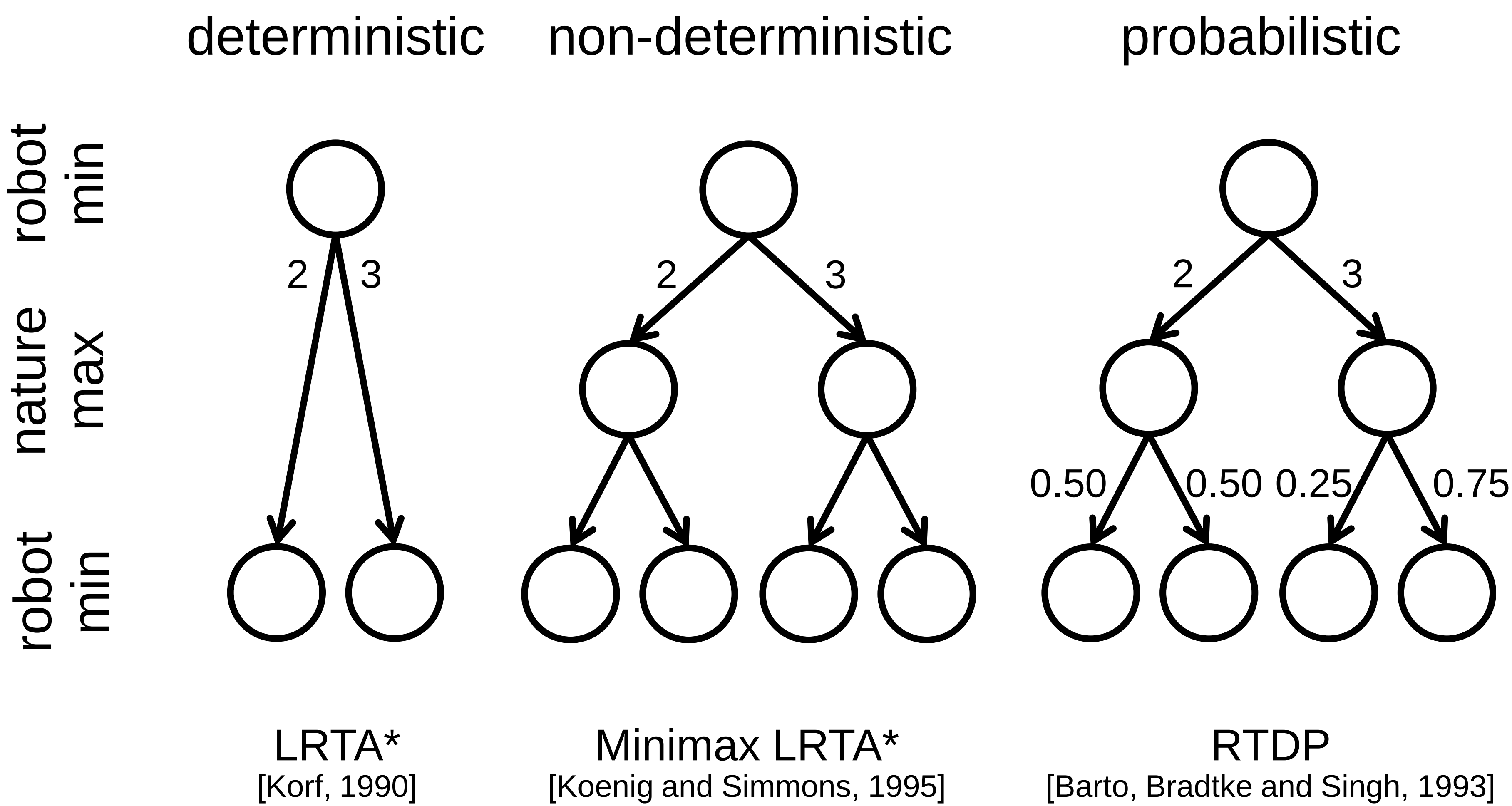
Real-Time Adaptive A* (RTAA*)

	RTAA*			LRTA*		
	expansions	move-ments	planning time per search [ms]	expansions	move-ments	planning time per search [ms]
1	248538	248538	0.20	248538	248538	0.27
9	104229	56708	2.01	87613	47291	2.80
17	85866	33853	4.37	79313	30470	6.25
25	89258	26338	6.86	82851	23270	10.23
33	96840	22022	9.41	92908	20016	14.31
41	105703	18629	11.99	102788	17274	18.50
49	117036	16638	14.46	113140	15398	22.67
57	128560	15367	16.83	125013	14285	26.69

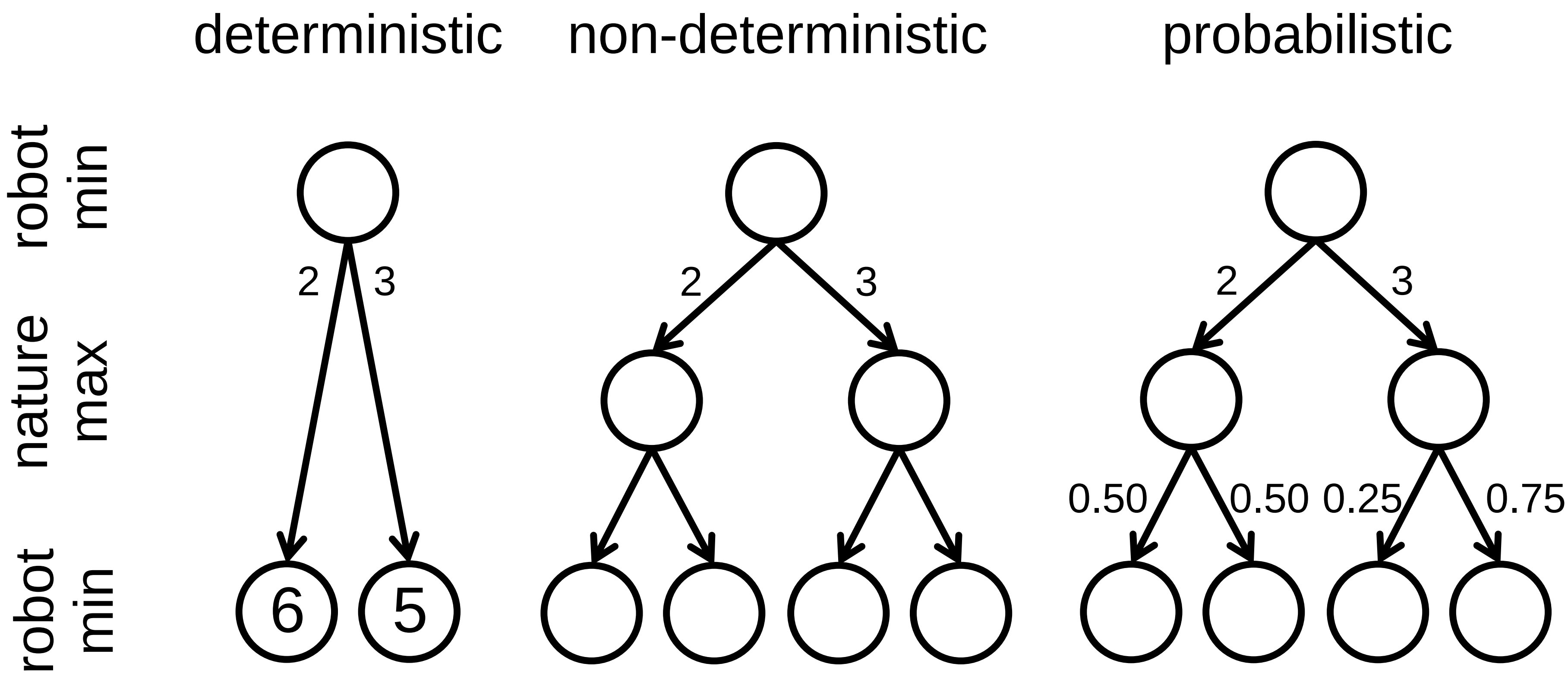
Real-Time Heuristic Search

- Learning-Real Time A* (LRTA*)
- Comparison of D* Lite and LRTA*
- Real-Time Adaptive A* (RTAA*)
- Generalizations of LRTA*: Minimax LRTA* and RTDP

Generalizations



Generalizations

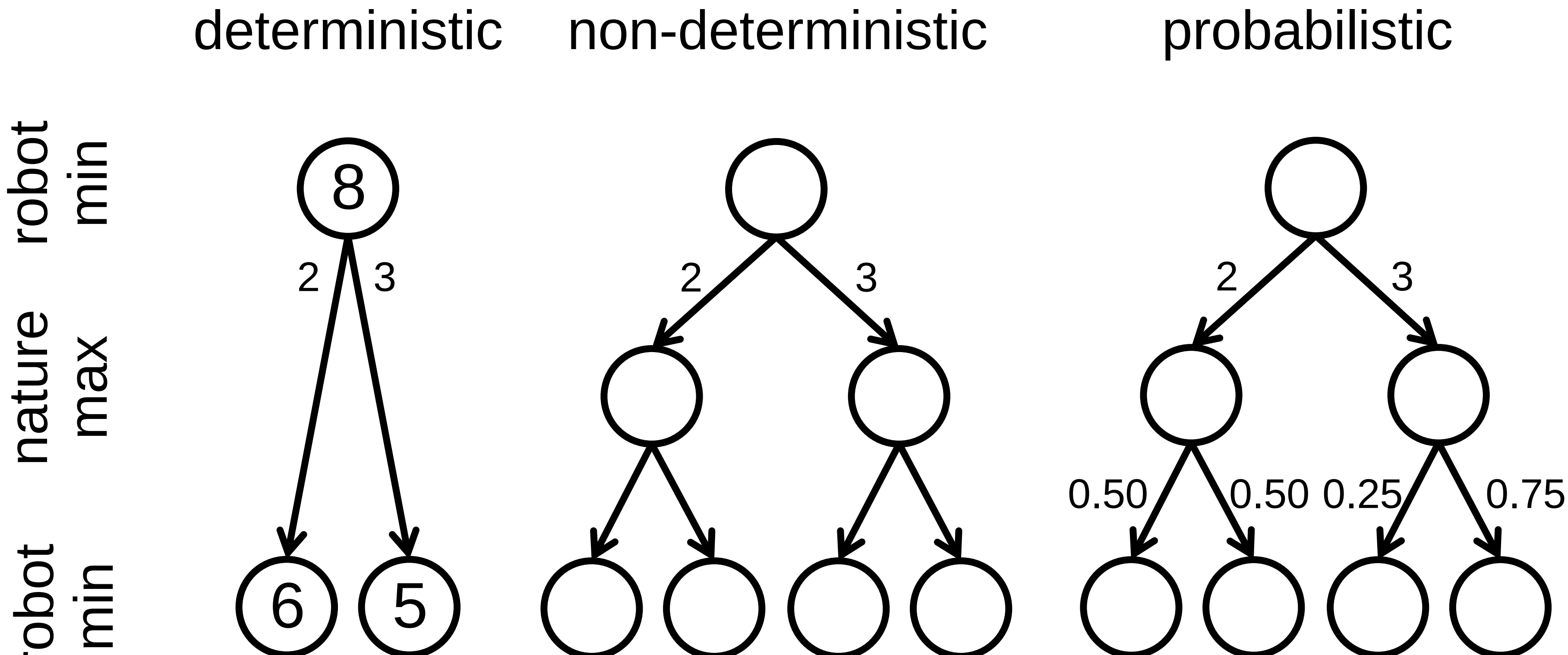


LRTA*
[Korf, 1990]

Minimax LRTA*
[Koenig and Simmons, 1995]

RTDP
[Barto, Bradtke and Singh, 1993]

Generalizations

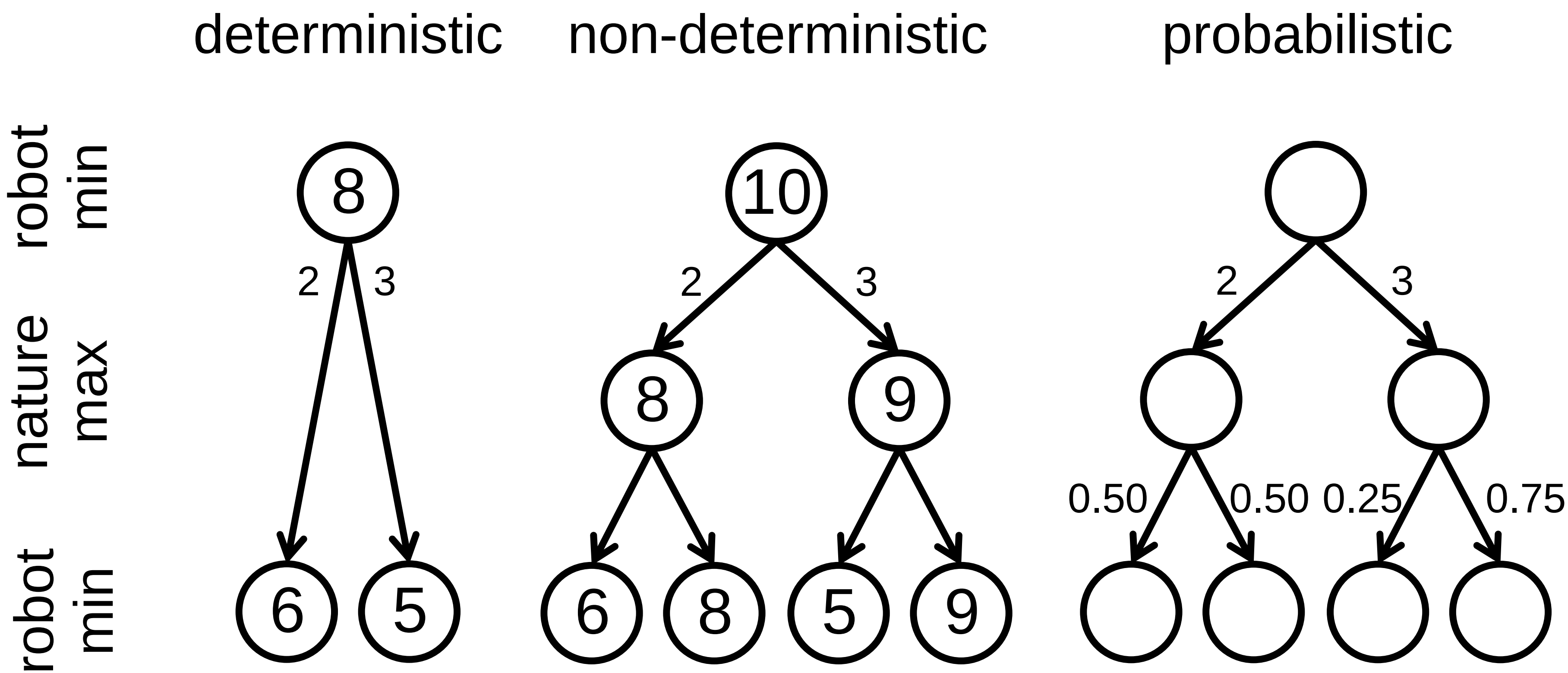


LRTA*
[Korf, 1990]

Minimax LRTA*
[Koenig and Simmons, 1995]

RTDP
[Barto, Bradtke and Singh, 1993]

Generalizations

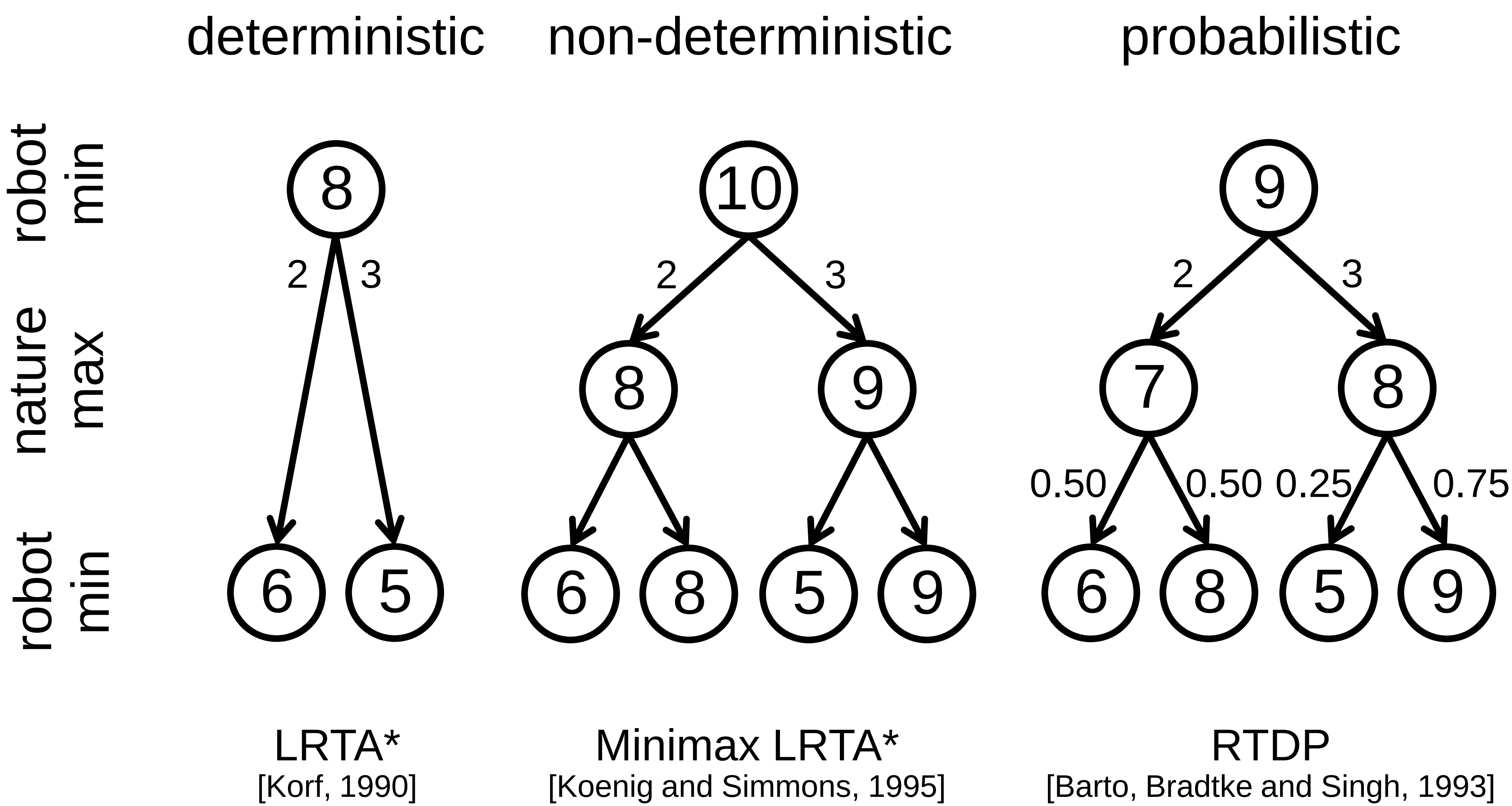


LRTA*
[Korf, 1990]

Minimax LRTA*
[Koenig and Simmons, 1995]

RTDP
[Barto, Bradtke and Singh, 1993]

Generalizations



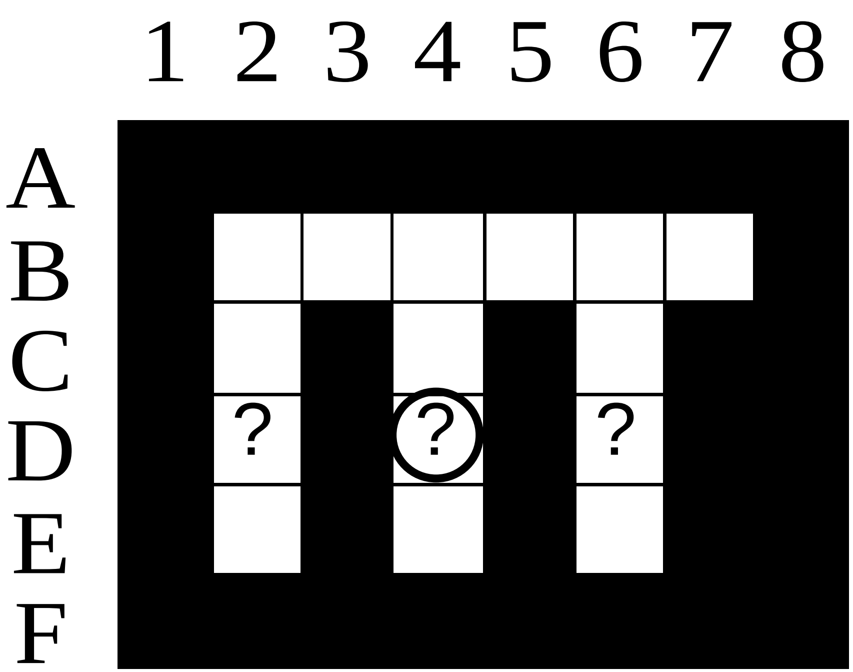
Generalizations

Properties of Learning Real-Time A* (LRTA*) [Korf, 1990]:

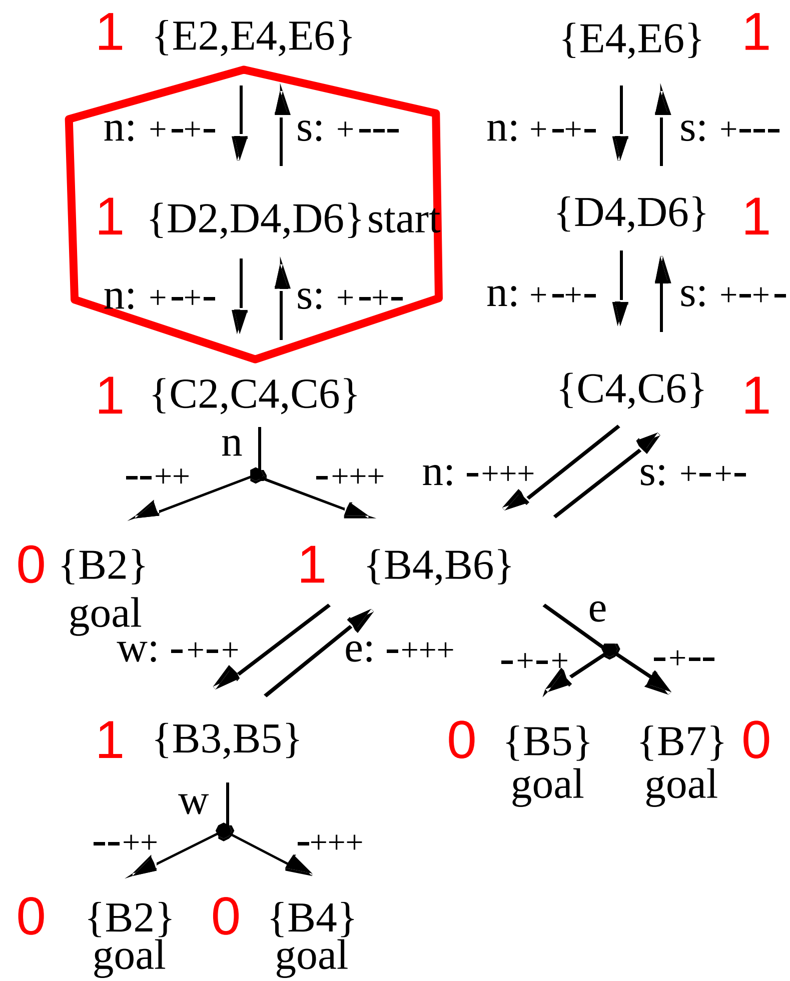
- The h-values of the same state are monotonically nondecreasing over time and thus indeed become more informed over time.
- The h-values remain consistent.
- The robot reaches the goal with $O|V|^2)$ movements in safely explorable state spaces [Koenig, 2001], where $|V|$ is the number of states (= unblocked cells).
- If the robot is reset into the start whenever it reaches the goal then the number of times that it does not follow a shortest path from the start to the goal is bounded from above by a constant if the cost increases are bounded from below by a positive constant.

Generalizations

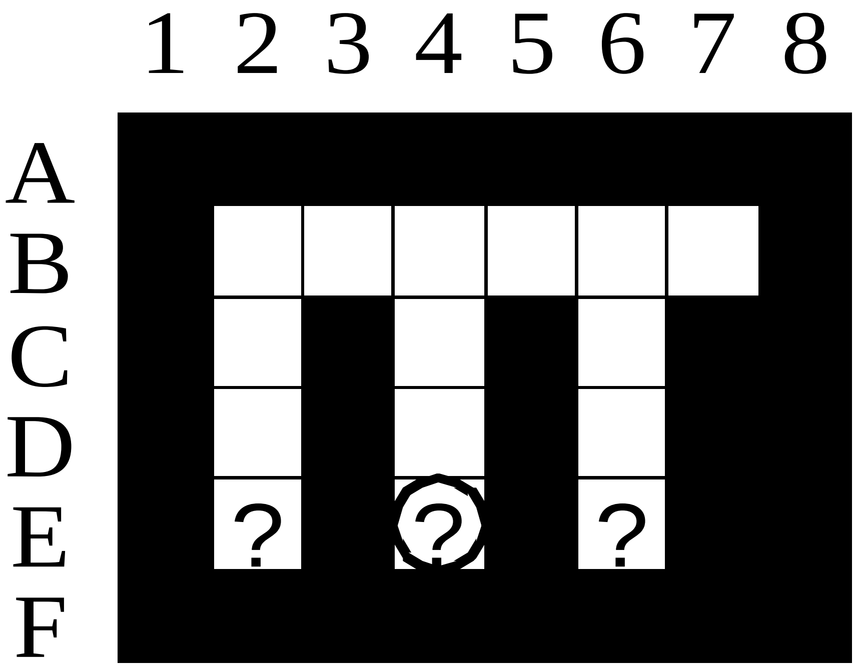
- Assume that the robot is told that it starts in D2, D4 or D6.

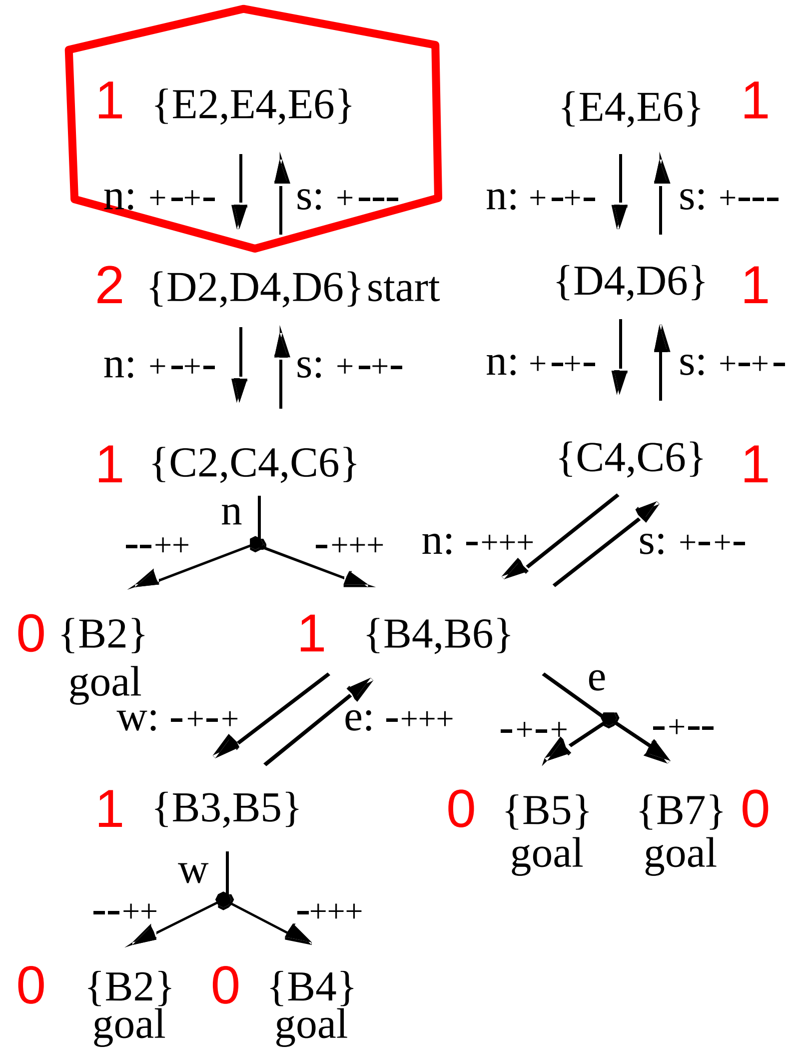


4-neighbor grid



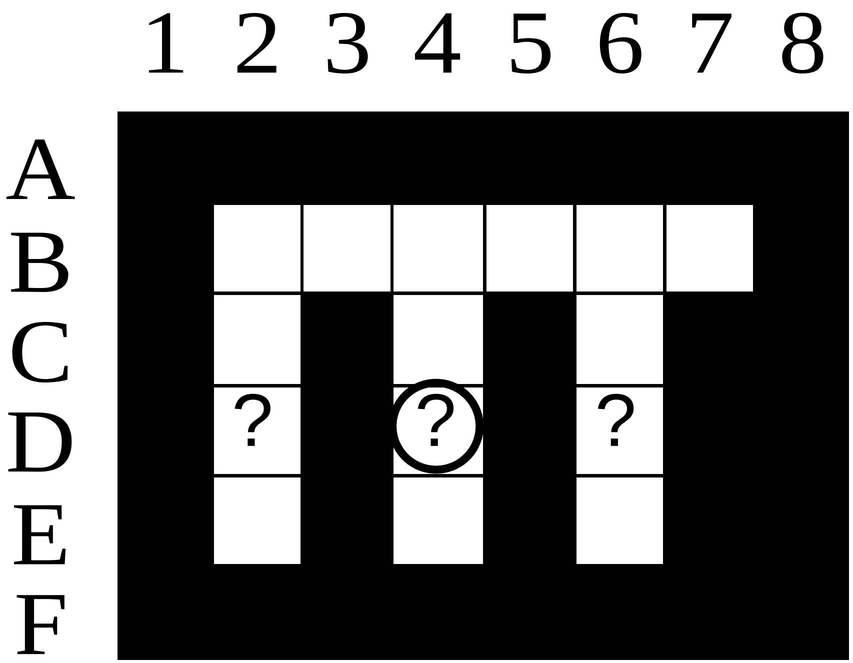
Generalizations



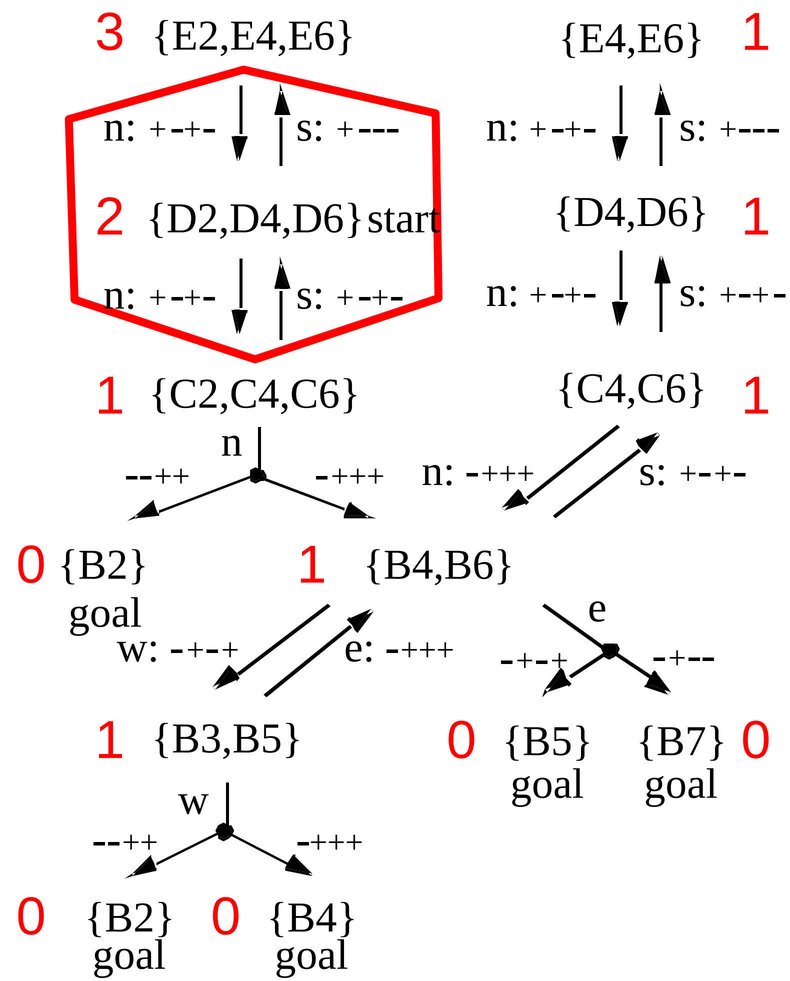


4-neighbor grid

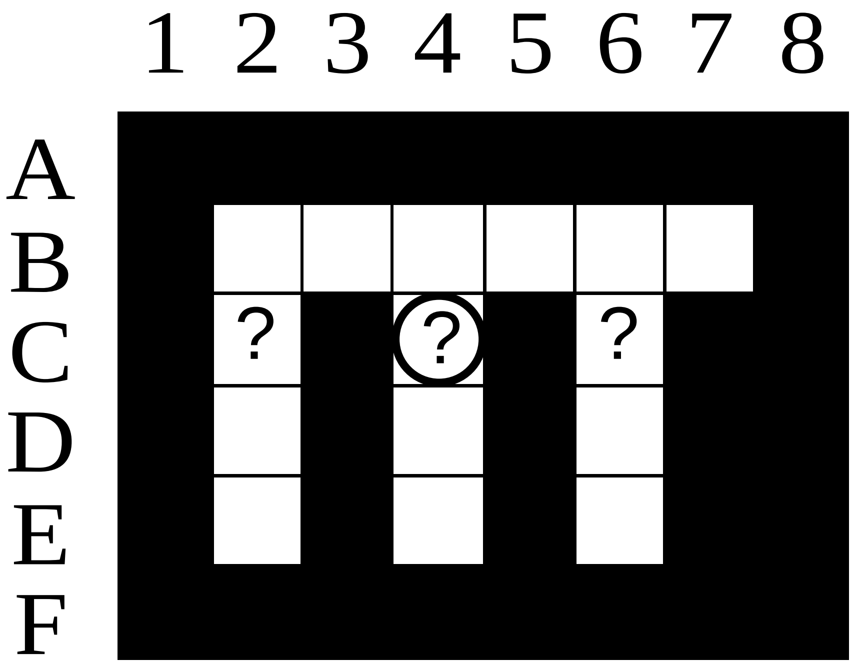
Generalizations

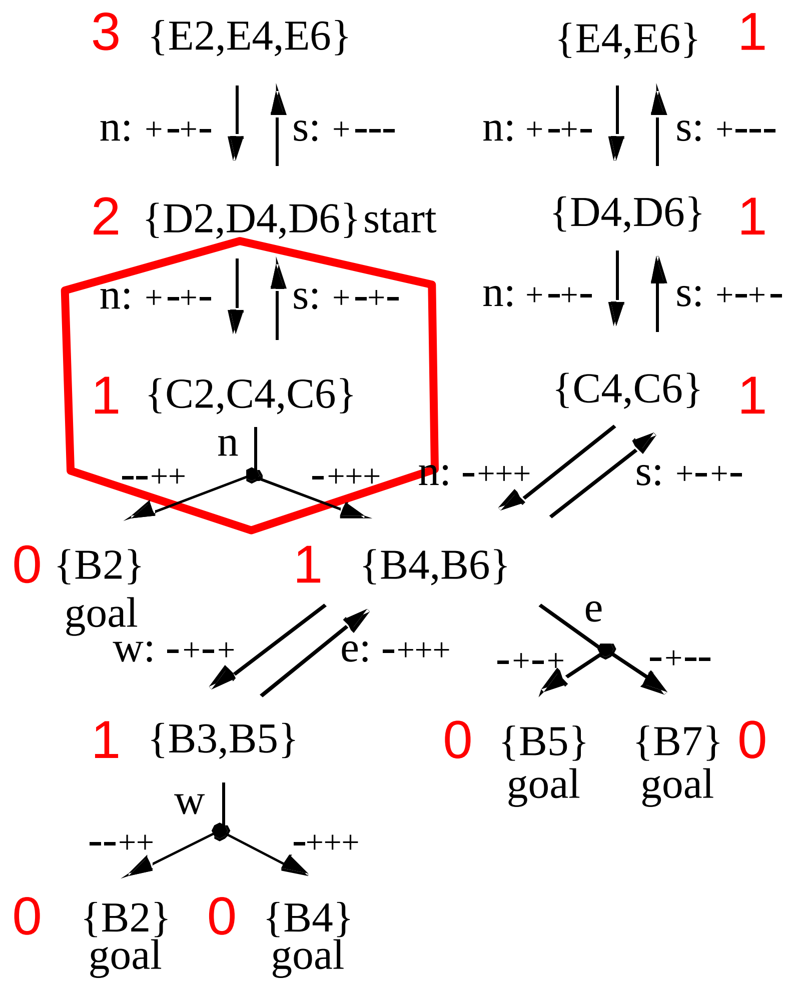


4-neighbor grid



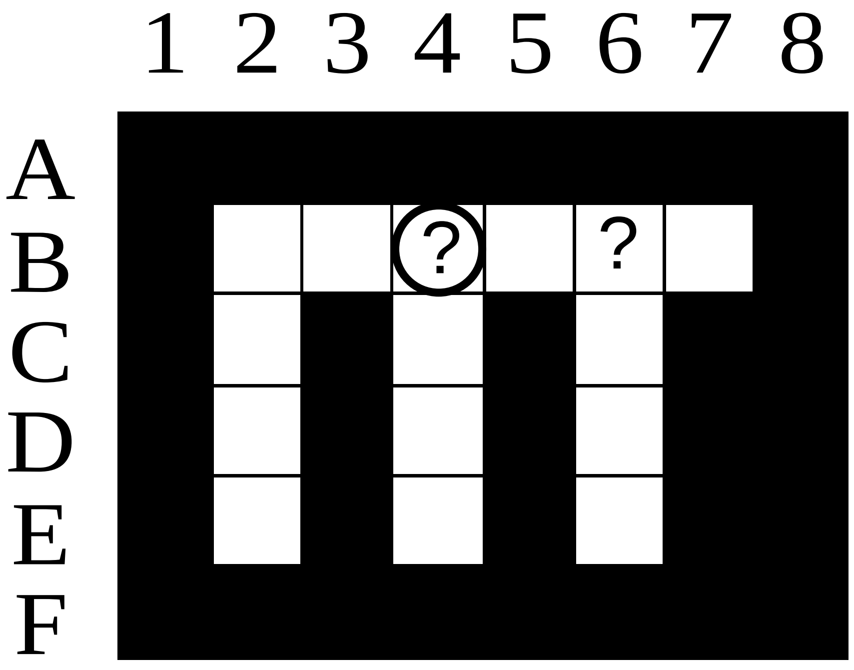
Generalizations



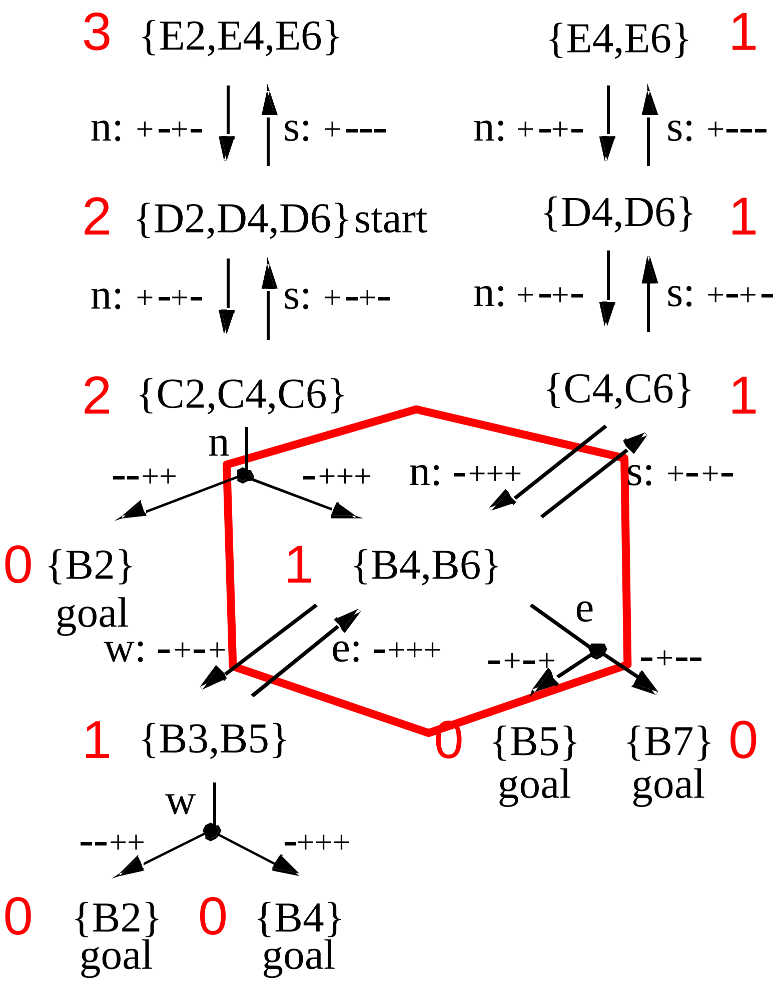


4-neighbor grid

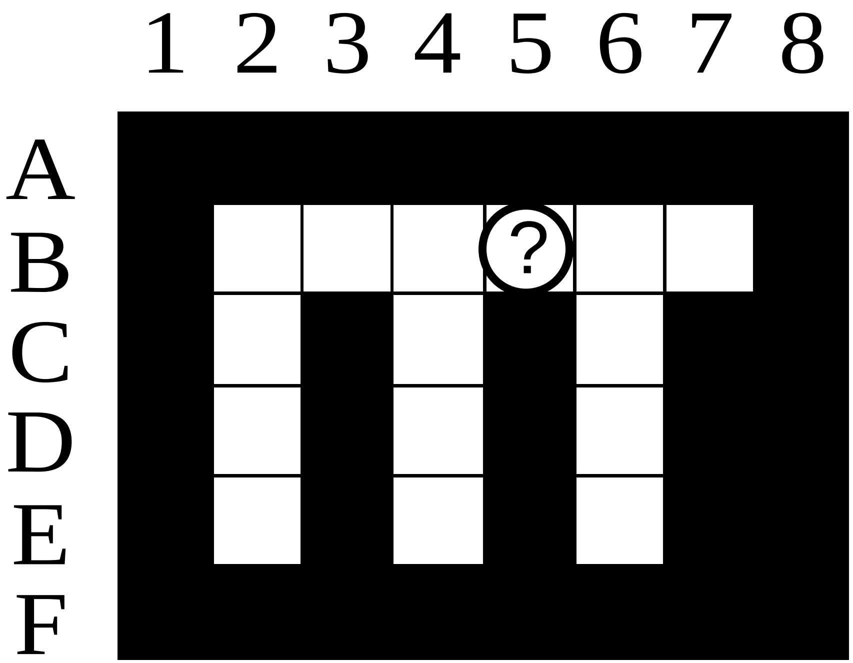
Generalizations



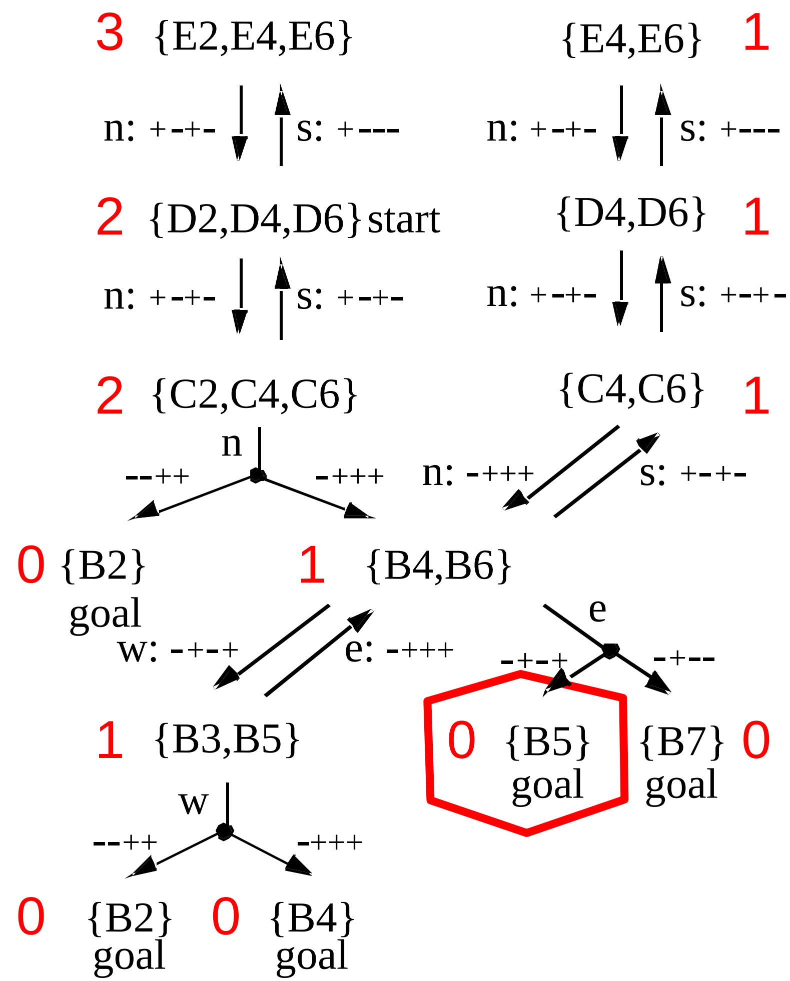
4-neighbor grid



Generalizations

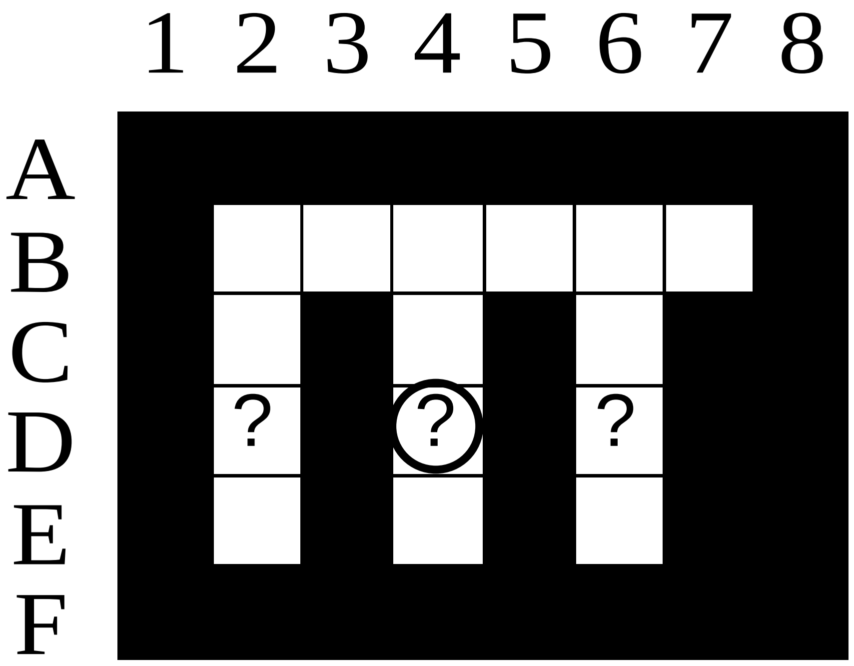


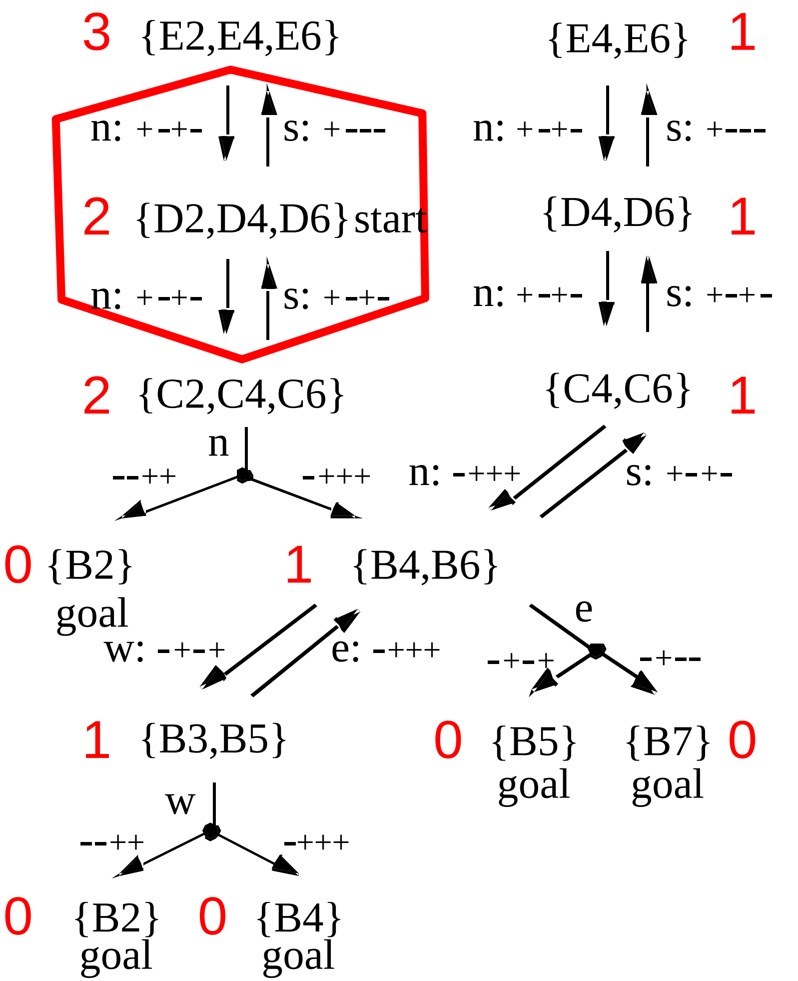
4-neighbor grid



Generalizations

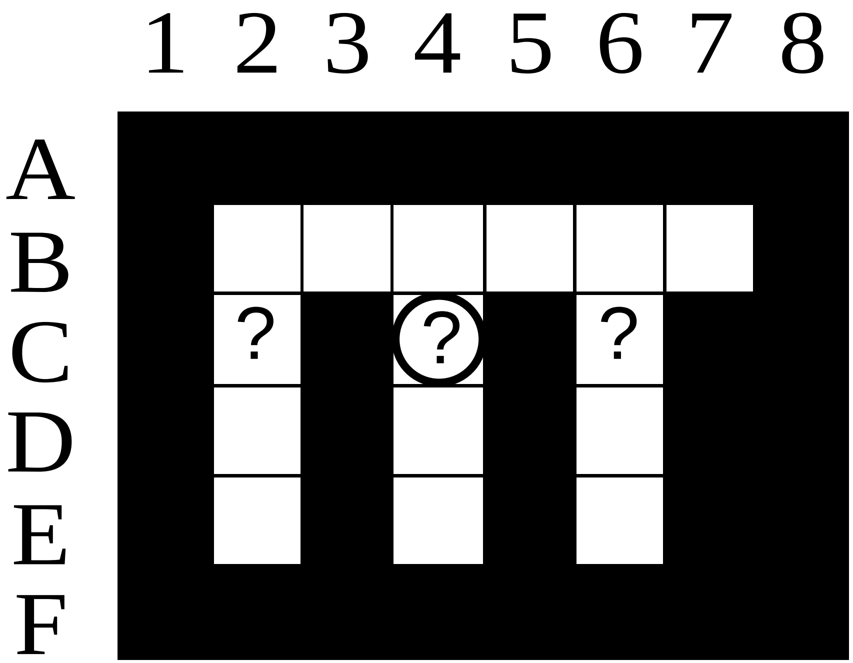
- Assume that the robot is told that it starts in D2, D4 or D6.



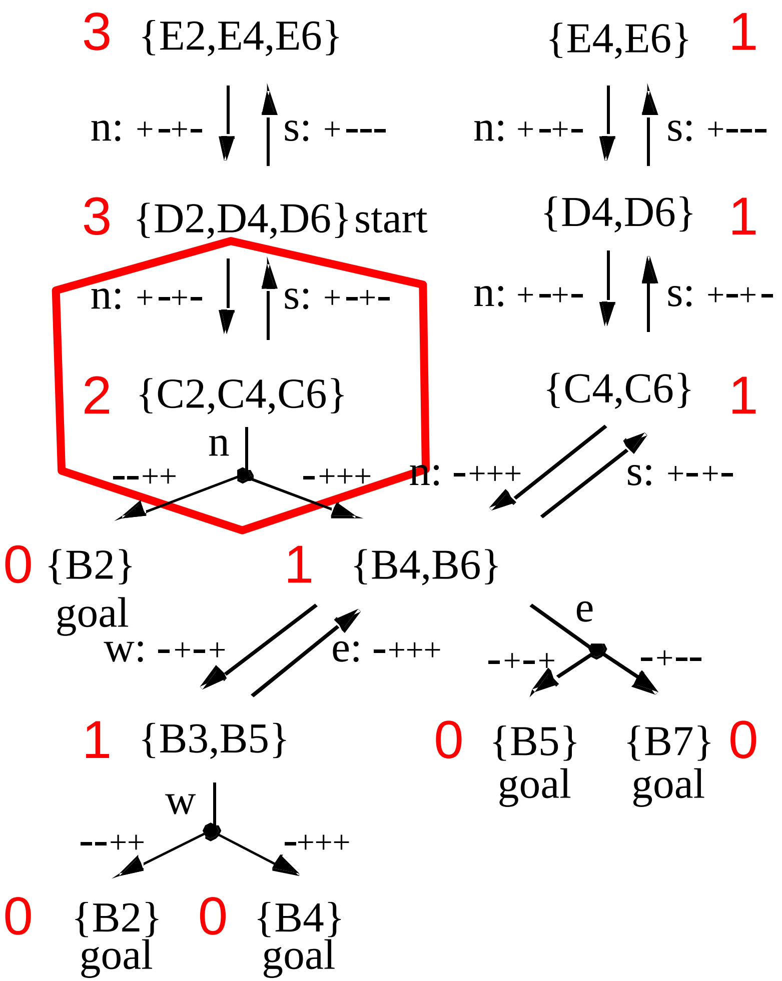


4-neighbor grid

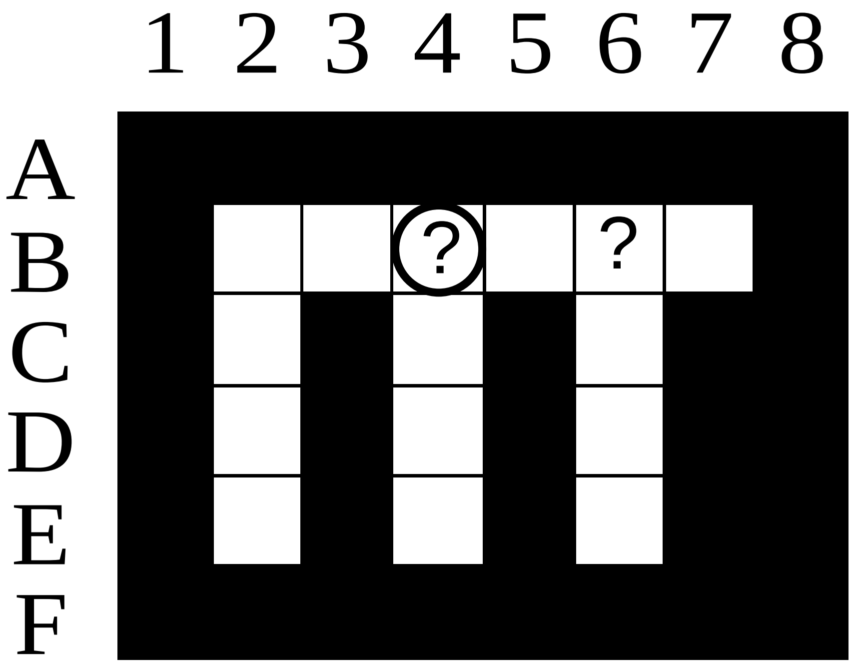
Generalizations



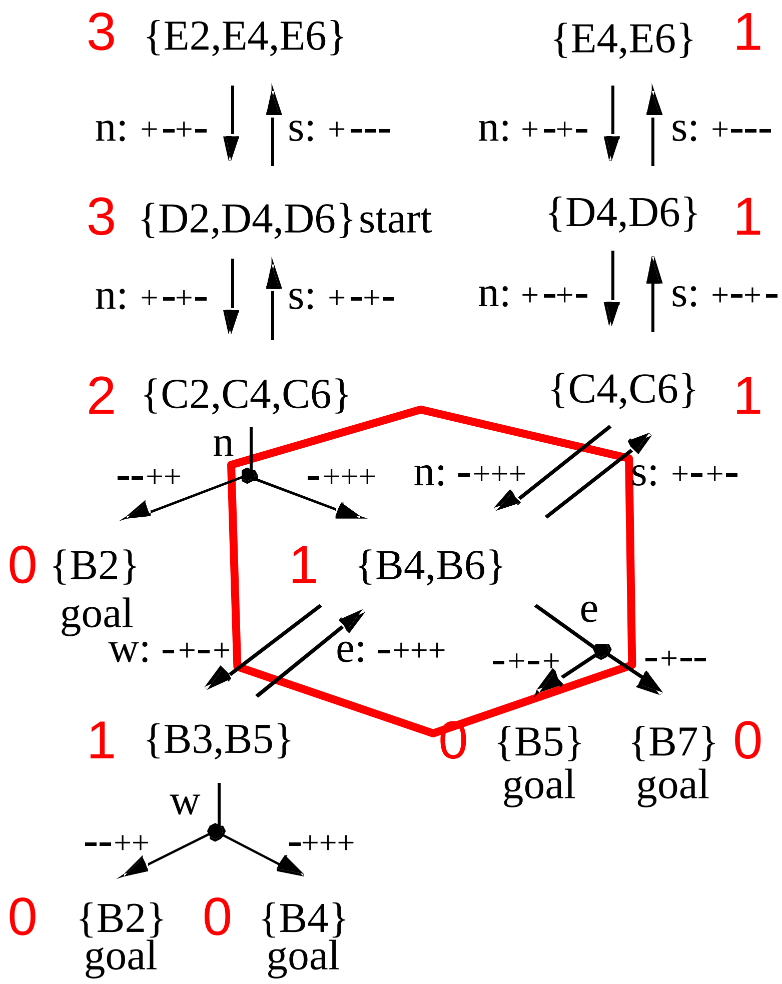
4-neighbor grid



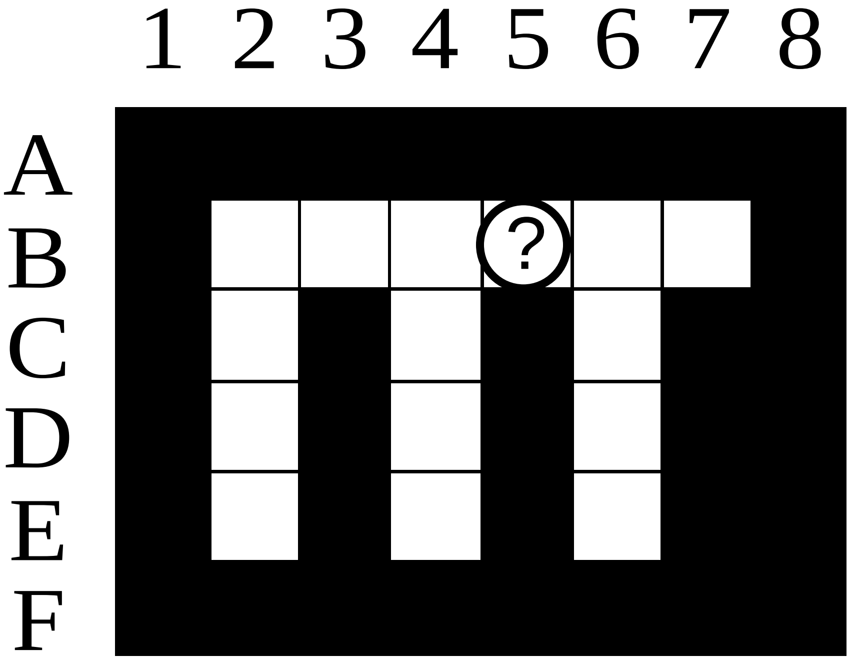
Generalizations

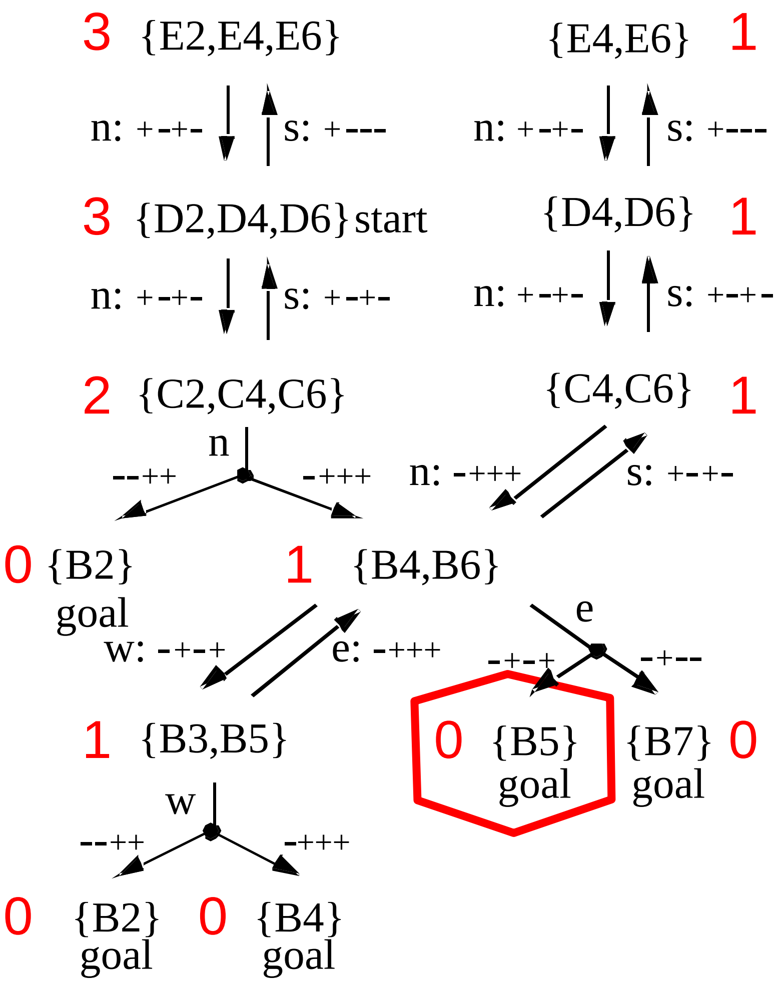


4-neighbor grid



Generalizations

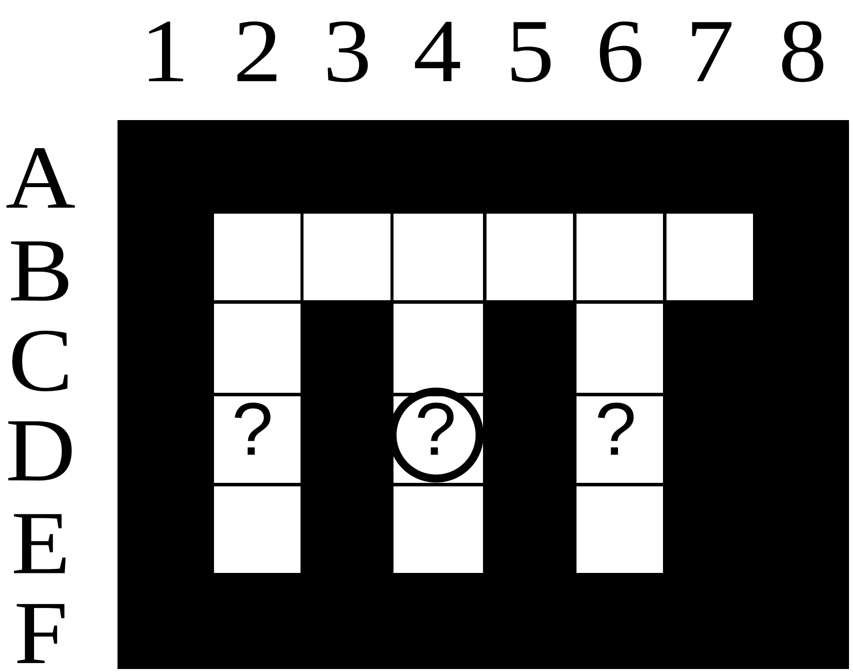




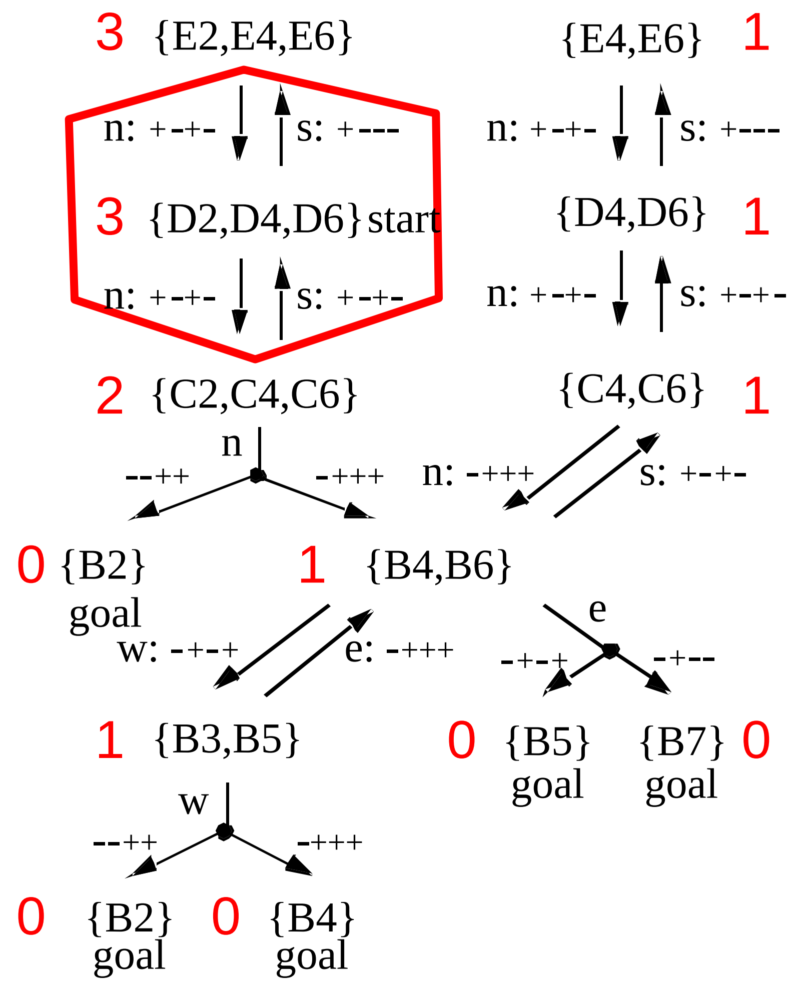
4-neighbor grid

Generalizations

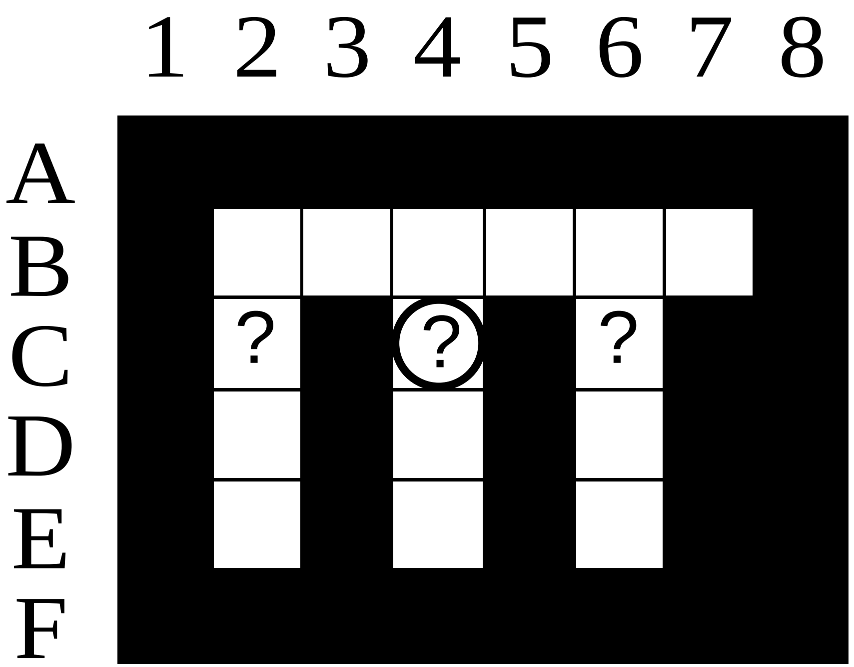
- Assume that the robot is told that it starts in D2, D4 or D6.



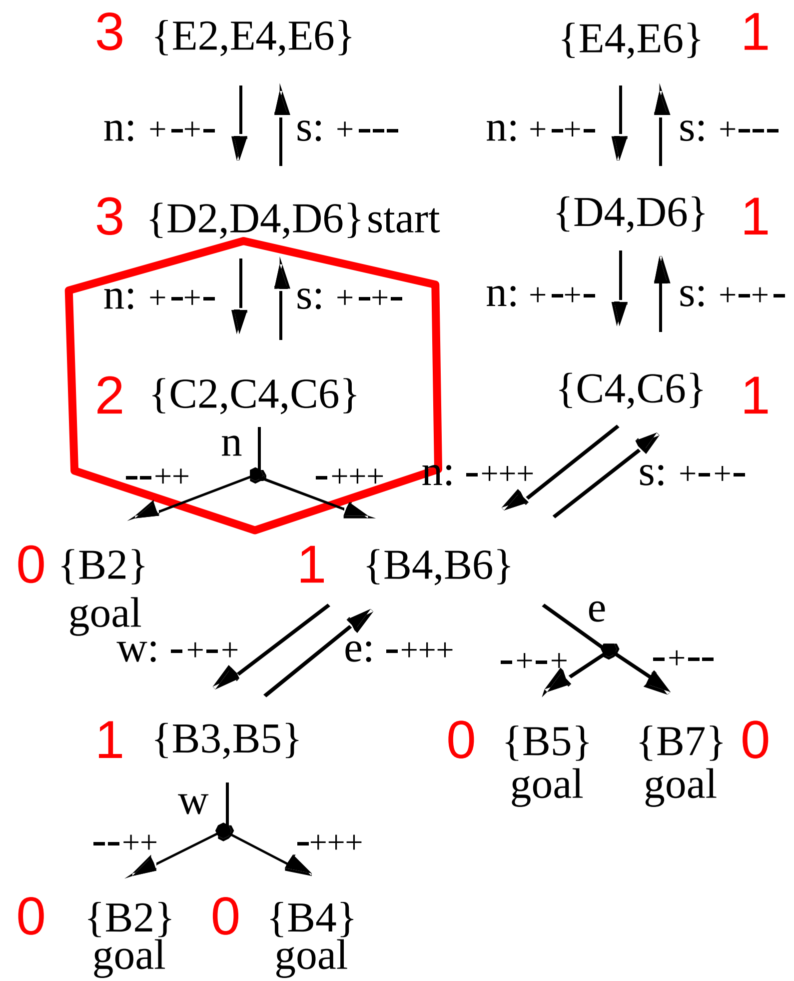
4-neighbor grid



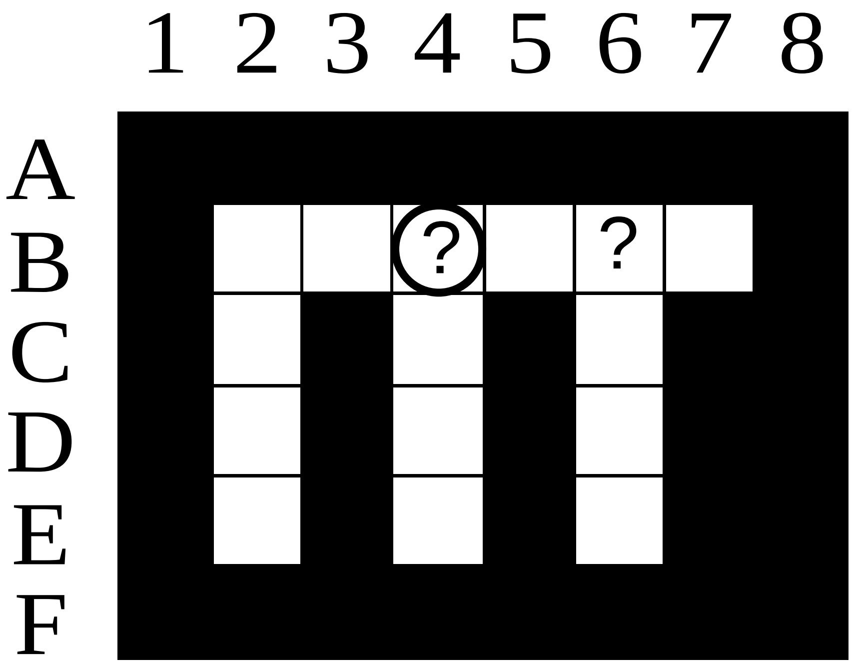
Generalizations

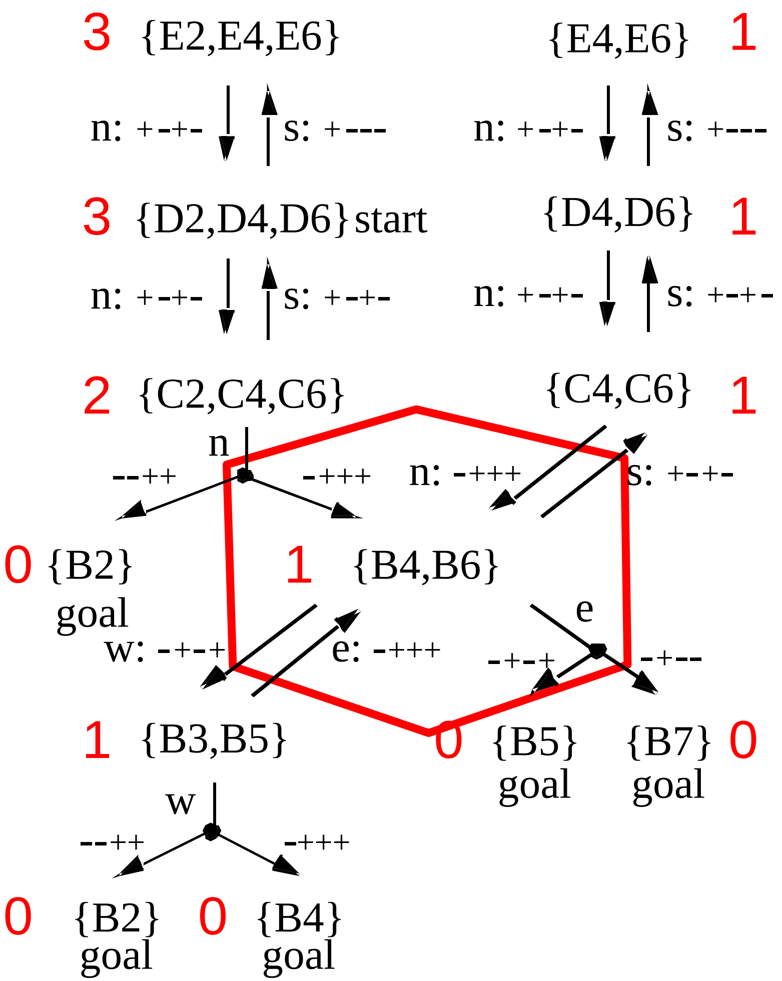


4-neighbor grid



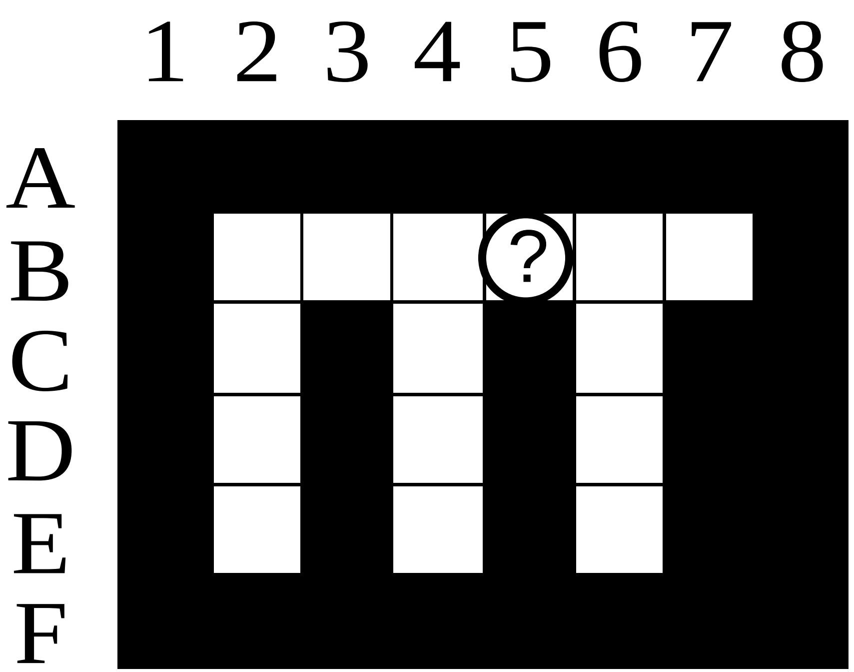
Generalizations





4-neighbor grid

Generalizations



4-neighbor grid

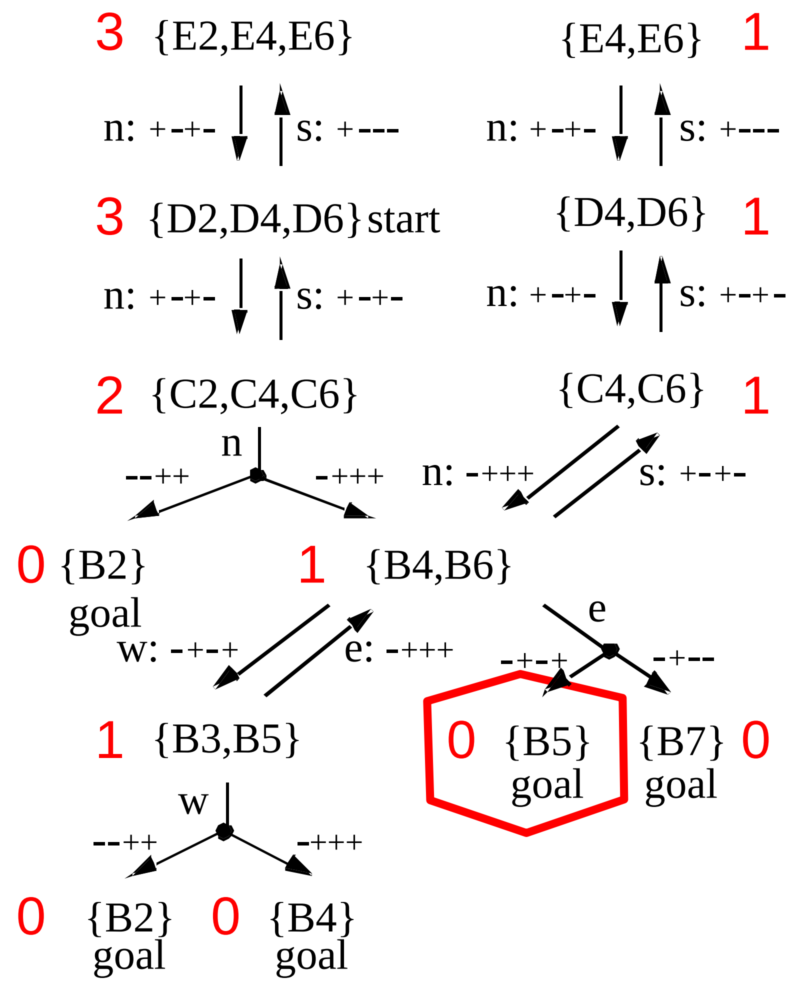


Table of Contents

- Modeling Planning Domains
  - Graphs, MDPs
- Planning Problems and Strategies
  - Localization, Mapping, Navigation in Unknown Terrain
  - Agent-Centered Search, Assumptive Planning
- Efficient Implementations of Planning Strategies
  - Incremental Heuristic Search

15 Minute Break

  - Real-Time Heuristic Search
  - Planning with Preferences on Uncertainty
  - Planning with Varying Abstractions

Planning with Incomplete Information

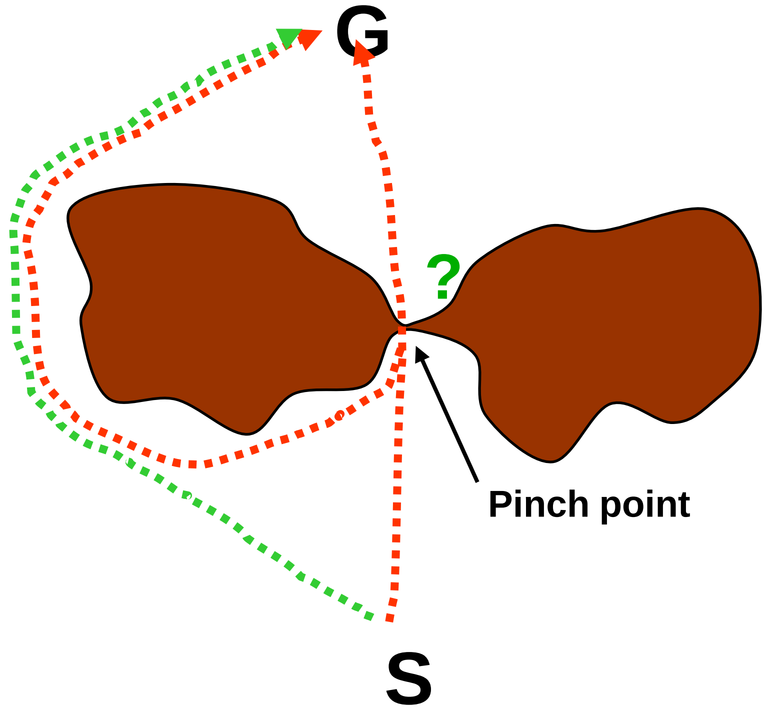
- planning with freespace assumption
  - fast deterministic planning
  - can make use of anytime/incremental/real-time implementations

  - but making assumptions can sometimes be highly suboptimal

Planning with Incomplete Information

- planning with freespace assumption
  - fast deterministic planning
  - can make use of anytime/incremental/real-time implementations

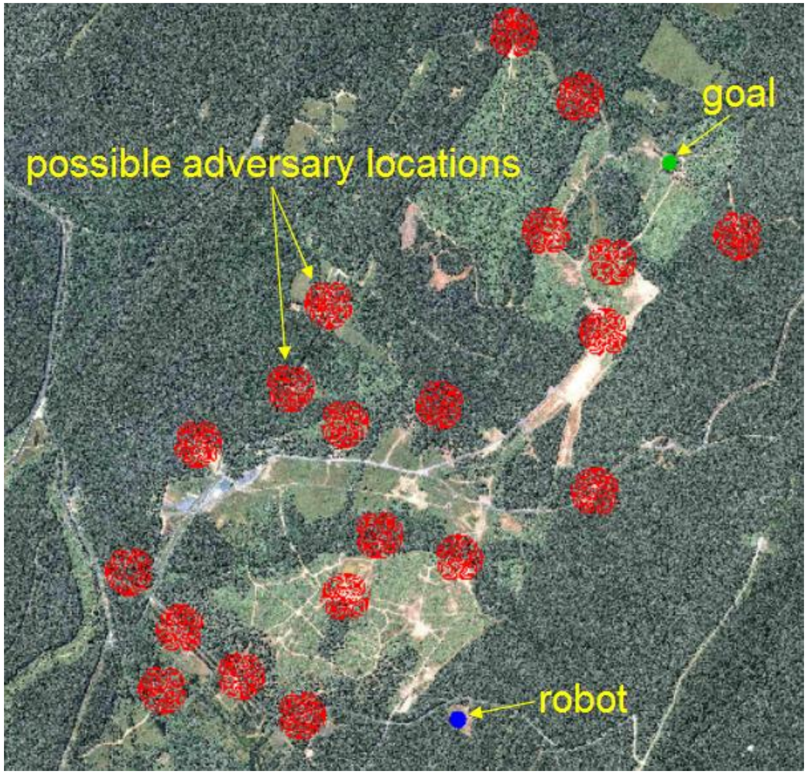
  - but making assumptions can sometimes be highly suboptimal



Path Clearance Problem

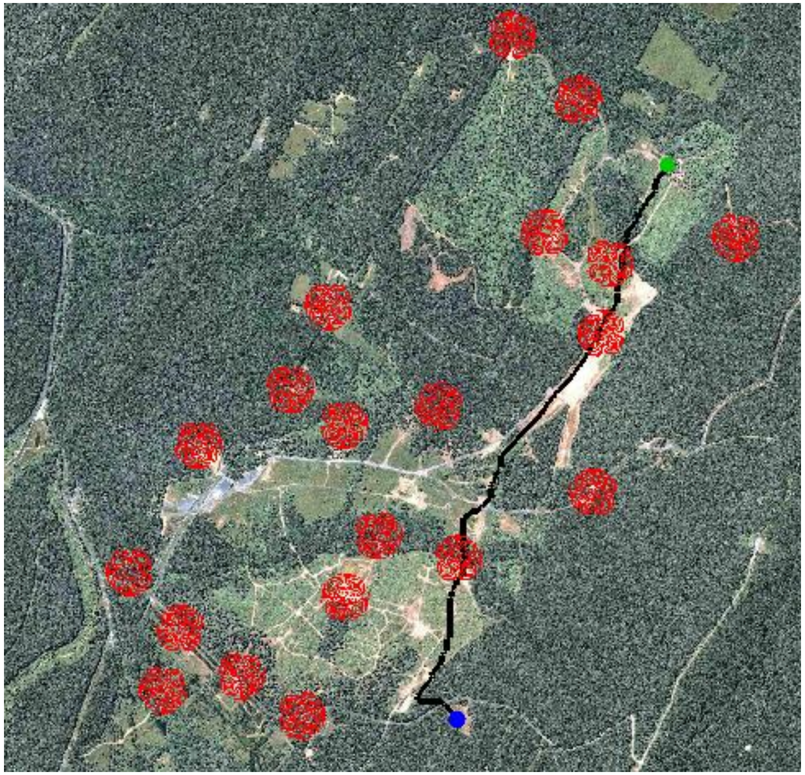
- quickly navigate to the goal without being detected by an adversary
- the robot can sense a possible adversary location at a distance
  - go through it if no adversary present
  - take a detour otherwise

environment size: 3.5km by 3.0km



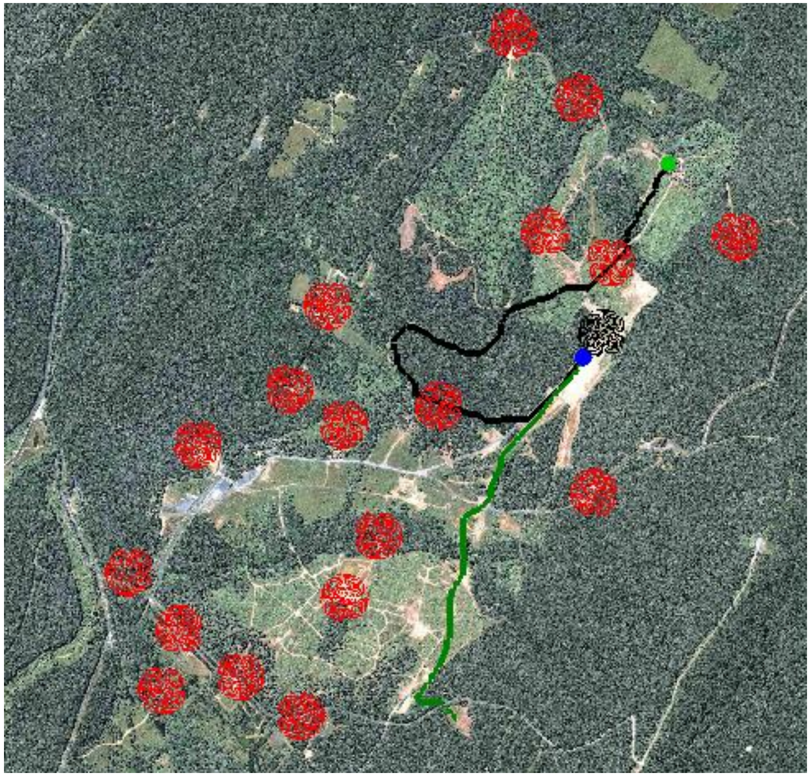
Path Clearance Problem

- planning problem: where to go + what to sense
- typical approaches to planning
  - assume no adversary present unless already detected
  - assign high cost to traversing possible adversary locations



Path Clearance Problem

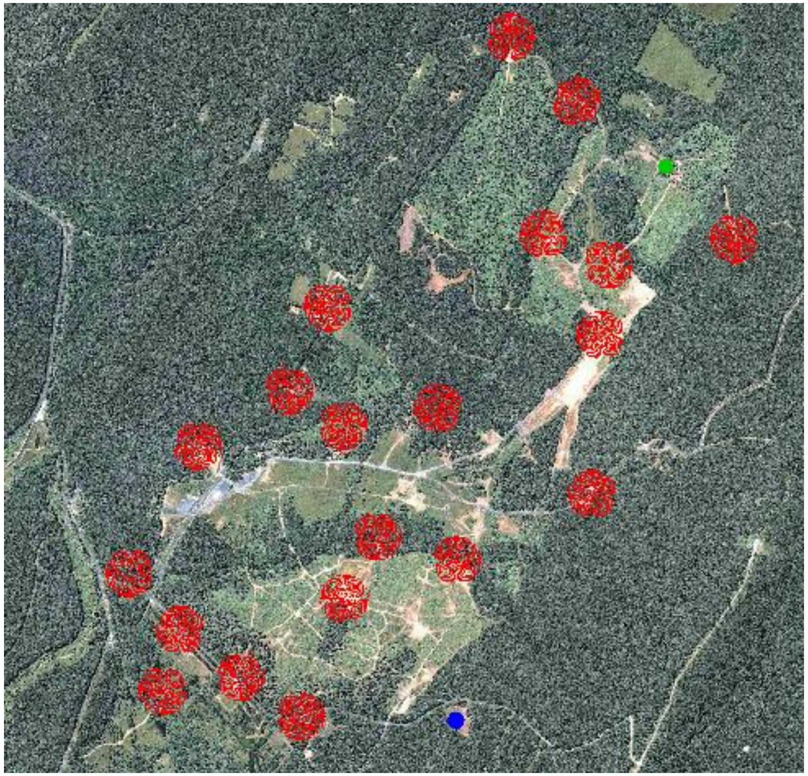
- planning problem: where to go + what to sense
- typical approaches to planning
  - assume no adversary present unless already detected
  - assign high cost to traversing possible adversary locations



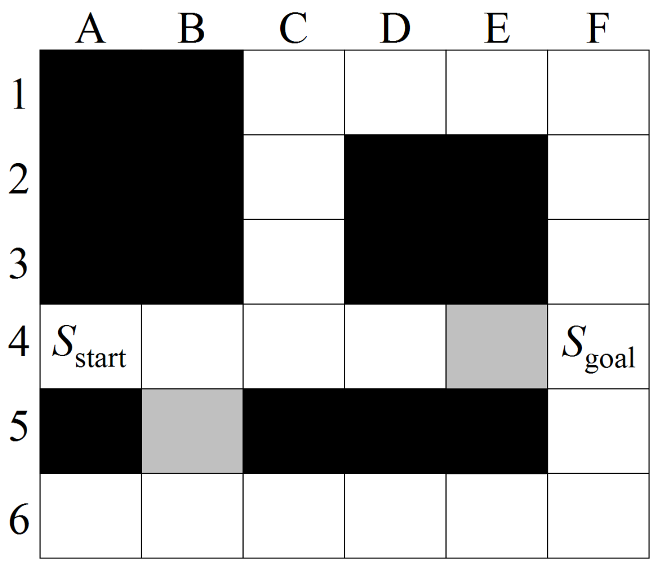
Path Clearance Problem

- probabilistic planning
  - minimizes the expected time/cost to goal
  - corresponds to planning with incomplete information
  - typically infeasible

size of belief state-space: $500*500*3^{20}$

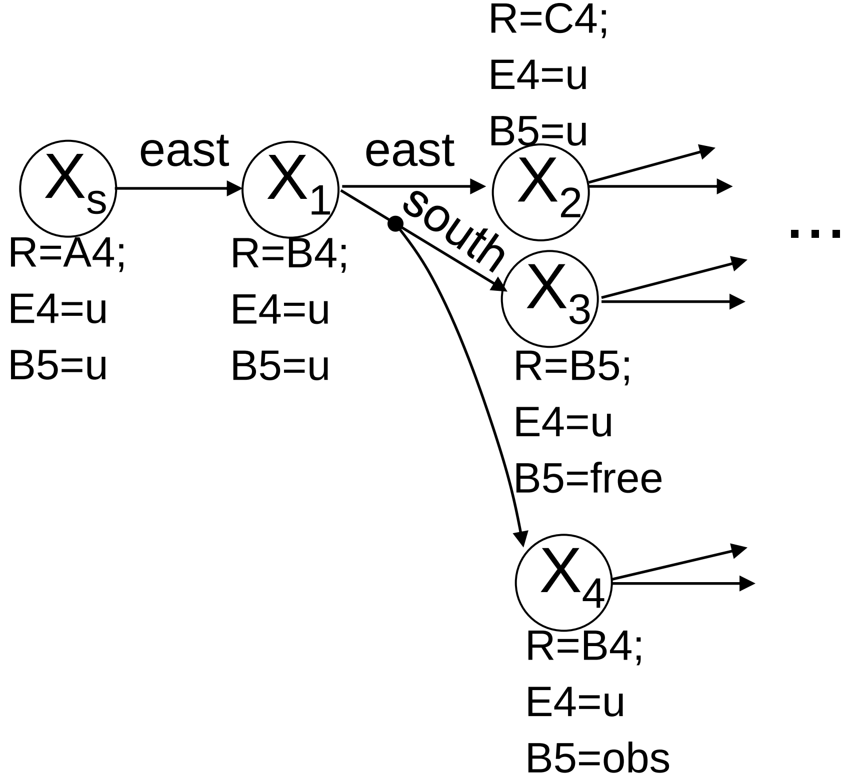


Planning with Incomplete Information



planning in belief state-spaces:

- exponential in the number of unknowns
- requires non-deterministic planning



Planning with Incomplete Information

can be solved efficiently by PPCP (Probabilistic Planning with Clear Preferences) [Likhachev & Stentz, AAAI'06] if:

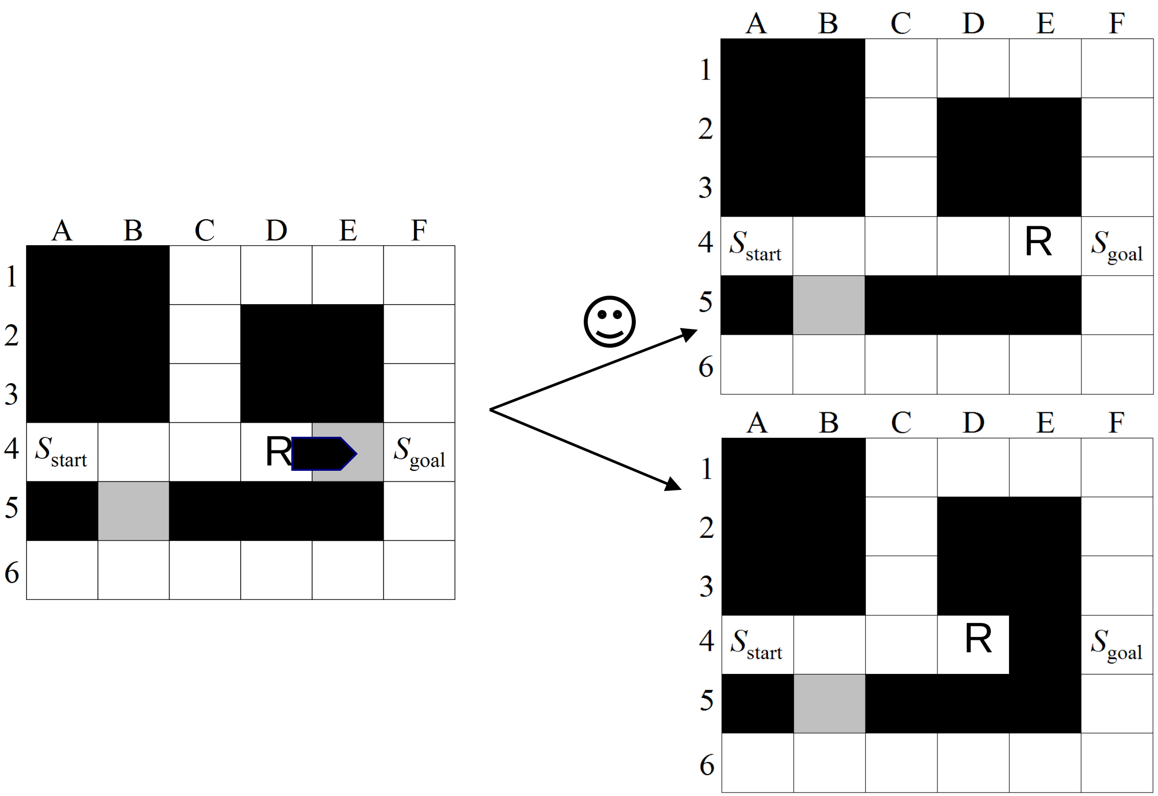
there exist *clear preferences on incomplete information*

Planning with Incomplete Information

can be solved efficiently by PPCP (Probabilistic Planning with Clear Preferences) [Likhachev & Stentz, AAAI'06] if:

there exist *clear preferences on incomplete information*

example of clearly preferred outcome of sensing (clear preference)

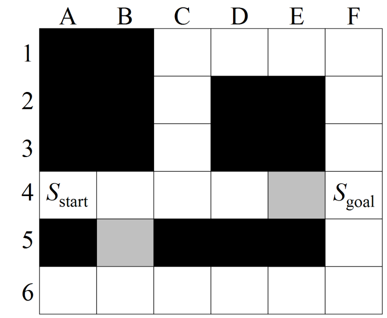


Formally, clear preference:

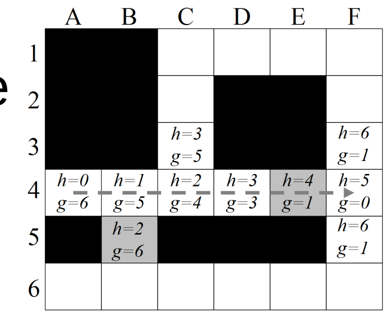
$\arg\min_{x'} (c(x,a,x') + v^*(x'))$,

where $v^*(x')$ – optimal exp. cost

PPCP [Likhachev & Stentz, AAAI'06]



- applies to an arbitrary graph (not just grid) with preferences on uncertainty in outcome/costs & perfect sensing
- solves the problem by running a series of A*-like searches

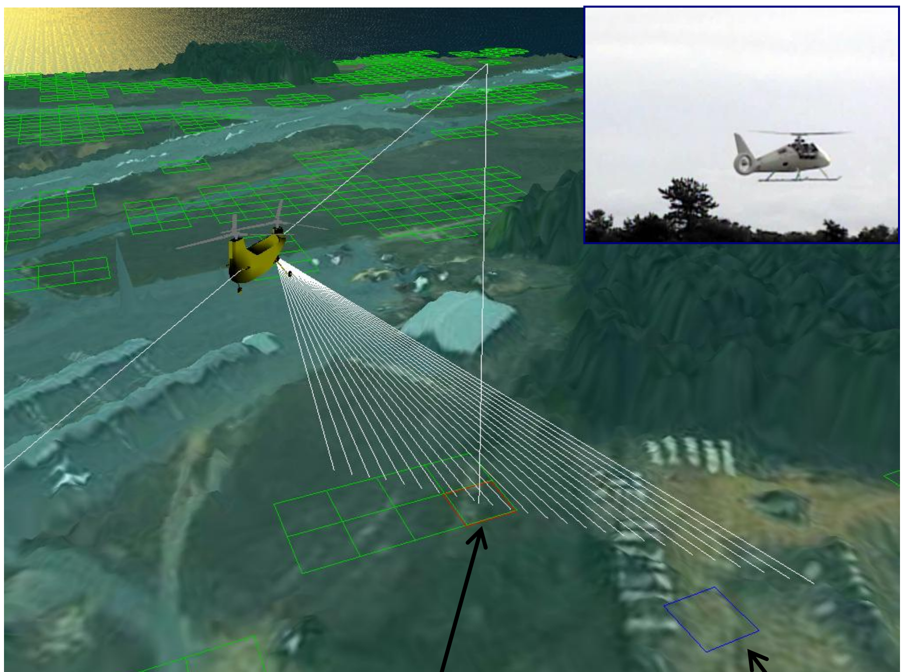


- each search is done on the original graph (e.g., 2D for navigation) whose size is exponentially smaller than the size of the belief state-space
  - as a result, scales to much larger problems and with much more uncertainty than if planning in the belief state-space directly
- converges to a solution that is optimal (minimizes the expected cost-to-goal) under certain conditions

Preferences on Incomplete Information

Landing site selection problem: where to go + what to sense

- land safely
- with minimum efforts
- as close to the desired goal as possible



closest to the goal landing site of desired condifence

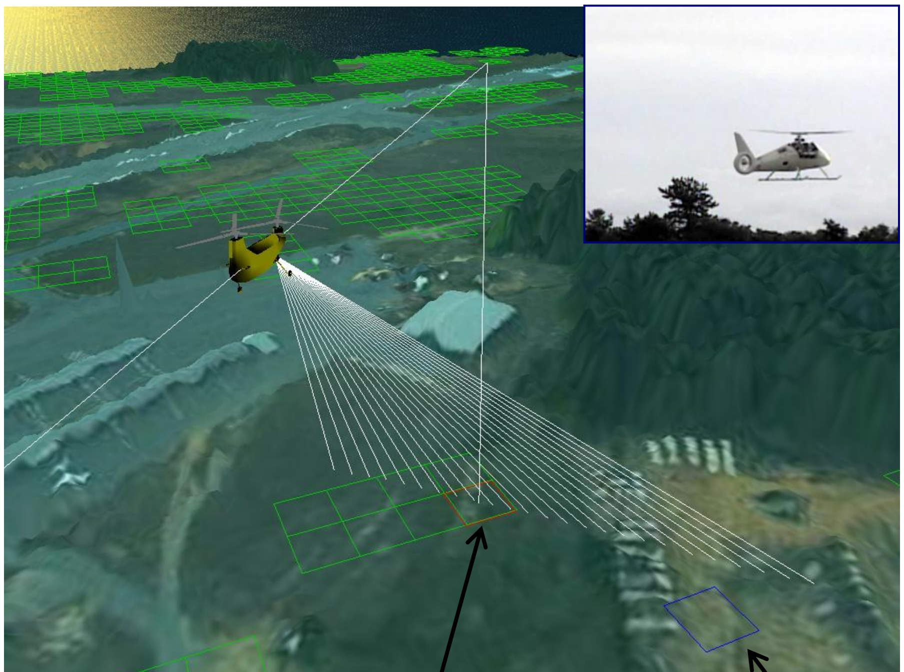
goal

Maxim

Preferences on Incomplete Information

Landing site selection problem: where to go + what to sense

- land safely
- with minimum efforts
- as close to the desired goal as possible

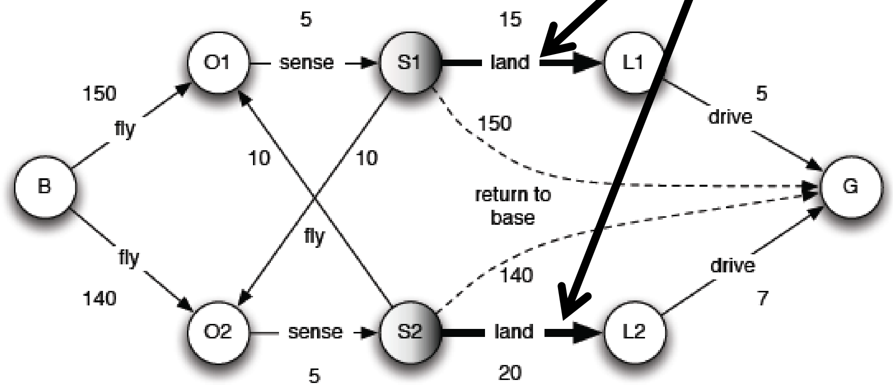


closest to the goal landing site of desired condifence

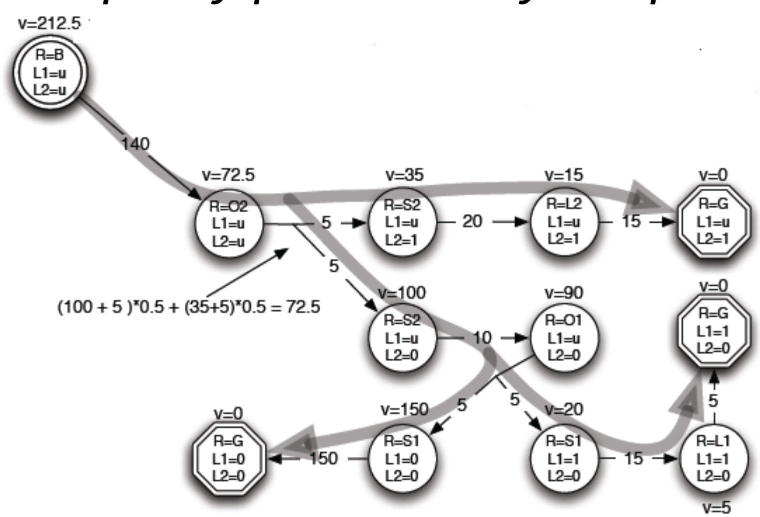
goal

unknown whether landing is possible

original graph



policy produced by the planner



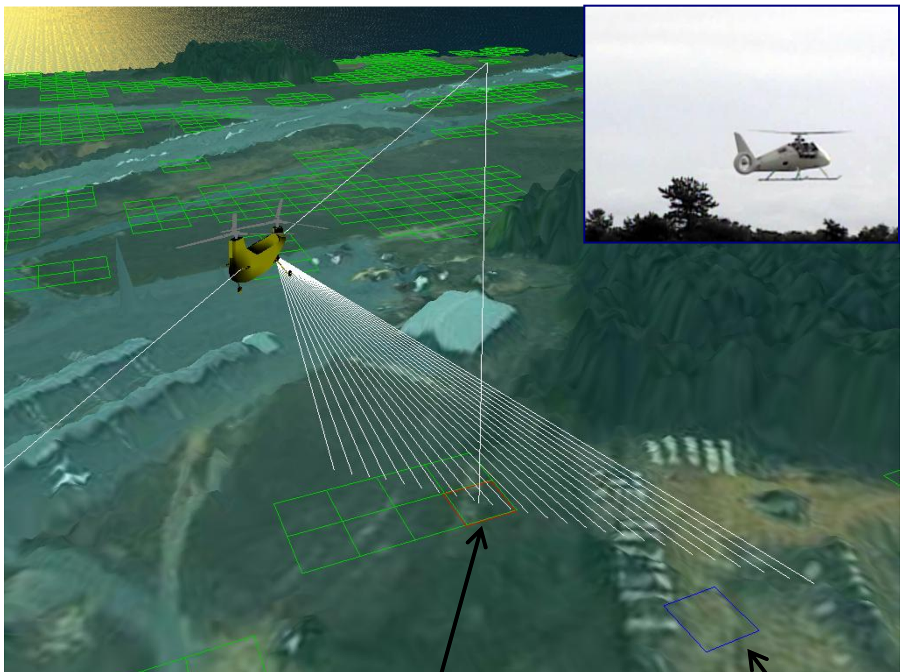
Maxim

Preferences on Incomplete Information

Landing site selection problem: where to go + what to sense

- land safely
- with minimum efforts
- as close to the desired goal as possible

PREFER TO HAVE GOOD LANDING SITE

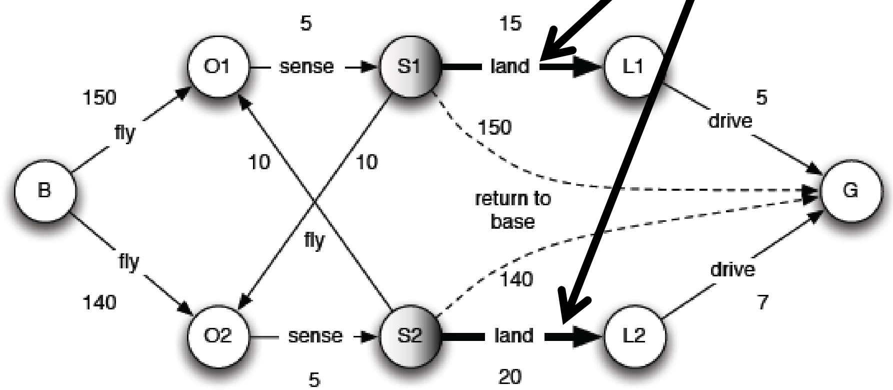


closest to the goal landing site of desired condifence

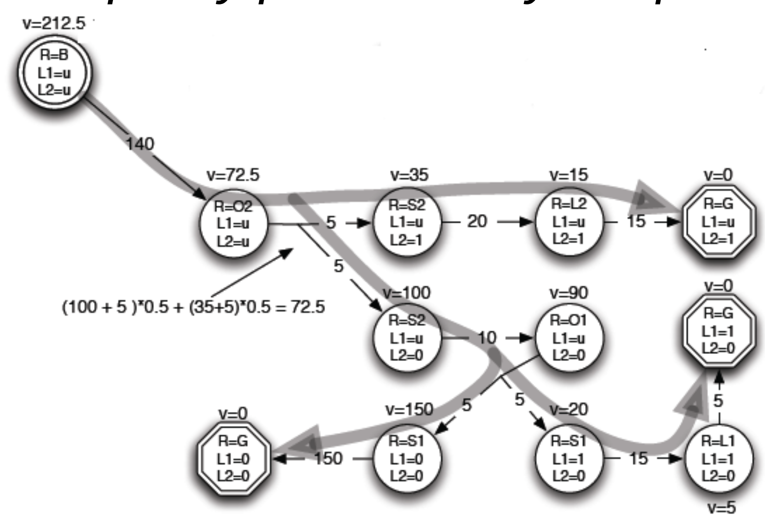
goal

unknown whether landing is possible

original graph



policy produced by the planner

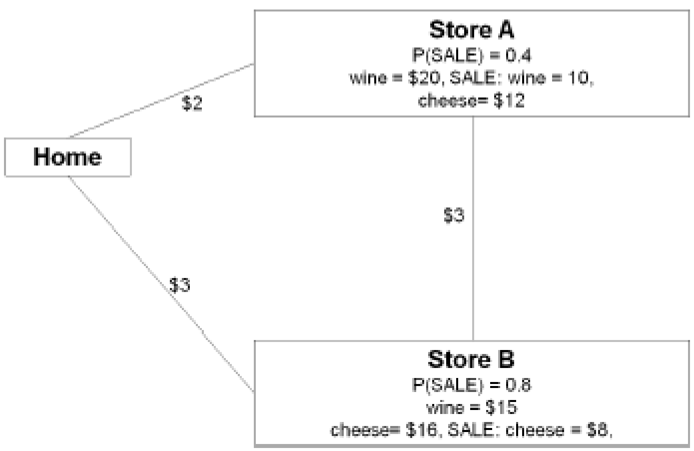


Preferences on Incomplete Information

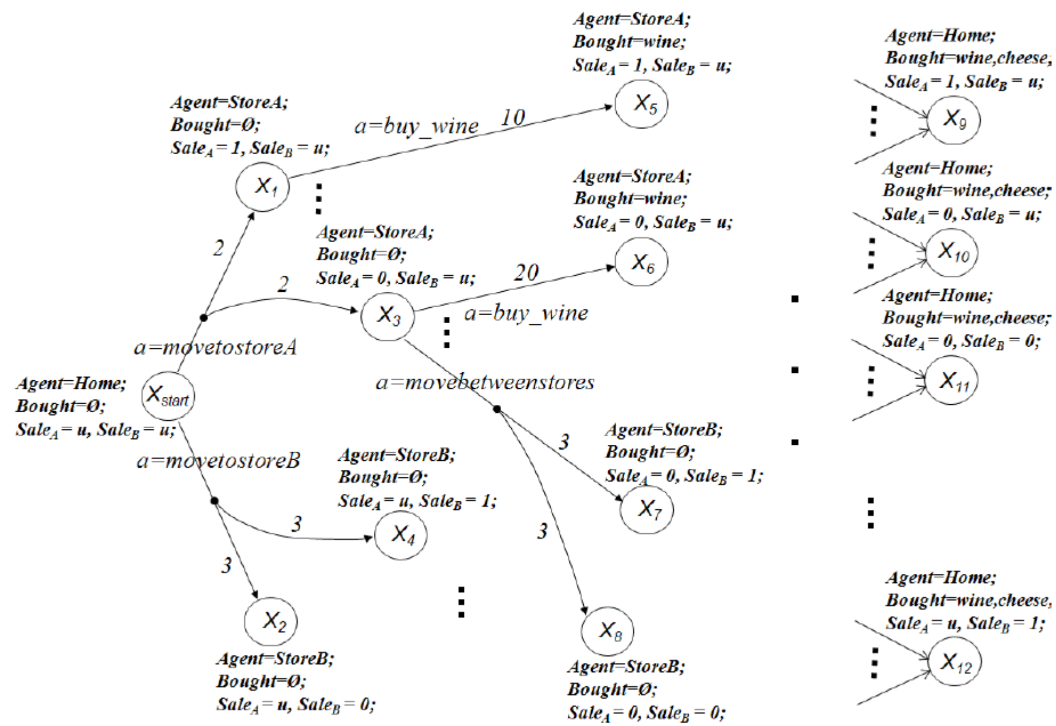
Grocery shopping under uncertainty in sale:

PREFER TO HAVE A SALE

original graph



belief state-space



Preferences on Incomplete Information

Examples of preferences on incomplete information:

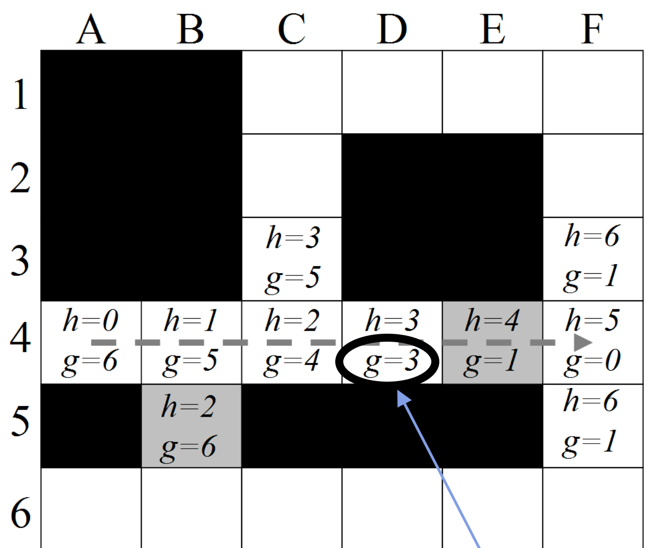
- Navigation in partially-known environments
- Route finding under uncertainty in traffic
- Air traffic management under uncertainty in weather conditions
- Grocery shopping under uncertainty in sale

…

Using Clear Preferences in PPCP

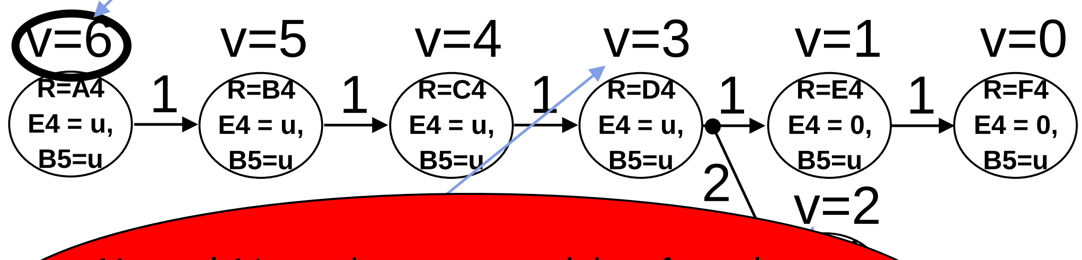
search backwards for a path from S_{start} to S_{goal}

assume: E4=u, B5=u



the policy after update:

the cost of the found path given by g(A4)



Normal A* requires monotonicity of g-values :
g(D4) = c(D4,east,E4=0) + g(E4) > g(E4)

clear preferences also provide monotonicity:
g(D4) > g(E4)
which makes A*-like search possible

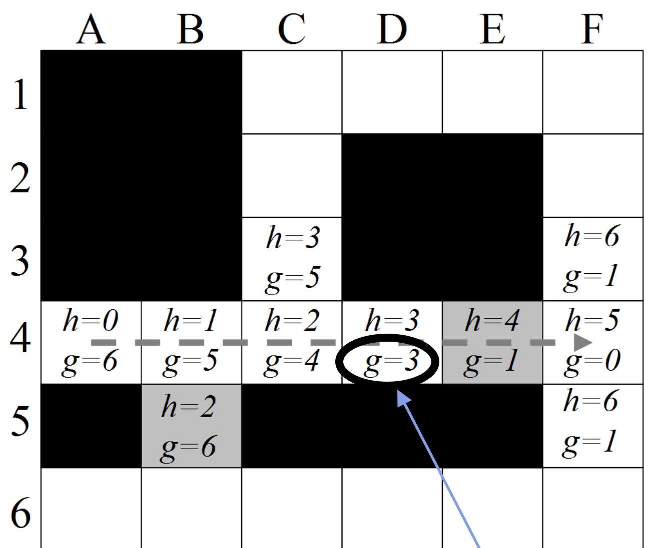
g(D4) = P(E4=0)(c(D4,east,E4=0)+g(E4)) + P(E4=1)(c(D4,east,E4=1)+v(D4,E4=1,B5=u)) = 0.5*(1+1)+0.5*(2+2) = 3

current estimate

Using Clear Preferences in PPCP

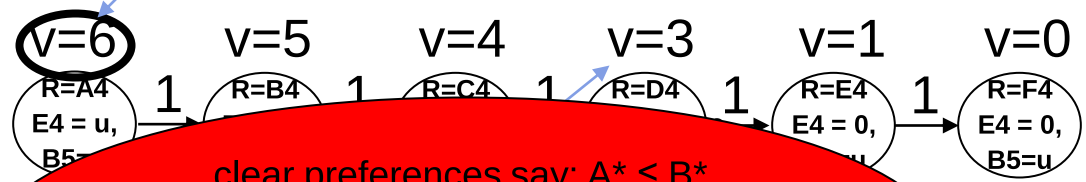
search backwards for a path from S_{start} to S_{goal}

assume: E4=u, B5=u



the policy after update:

the cost of the found path given by g(A4)



clear preferences say: A* ≤ B*
therefore we can set B = max(A,B)
A ≤ B
g(D4) = P(A) ·A + P(B)·B ≥ A > g(E4)

clear preferences also provide monotonicity:
g(D4) > g(E4)
which makes A*-like search possible

A B

g(D4) = P(E4=0)(c(D4,east,E4=0)+g(E4)) +
P(E4=1)(c(D4,east,E4=1)+v(D4,E4=1,B5=u)) =
0.5*(1+1)+0.5*(2+2) = 3

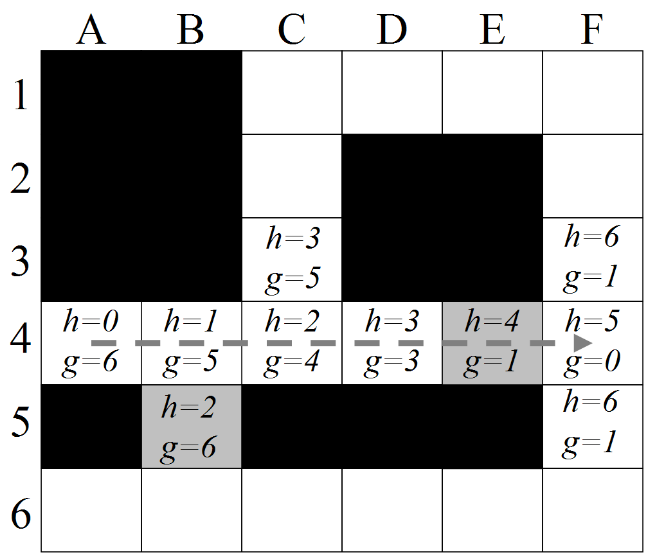
current estimate

Maxim

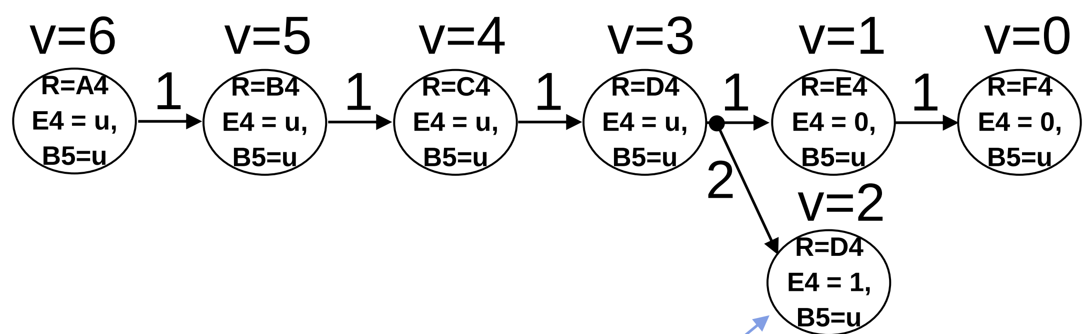
Run of PPCP

search backwards for a path from S_{start} to S_{goal}

assume: E4=u, B5=u



the policy after update:



PPCP repeatedly computes paths to states with negative Bellman error on the current policy until none left

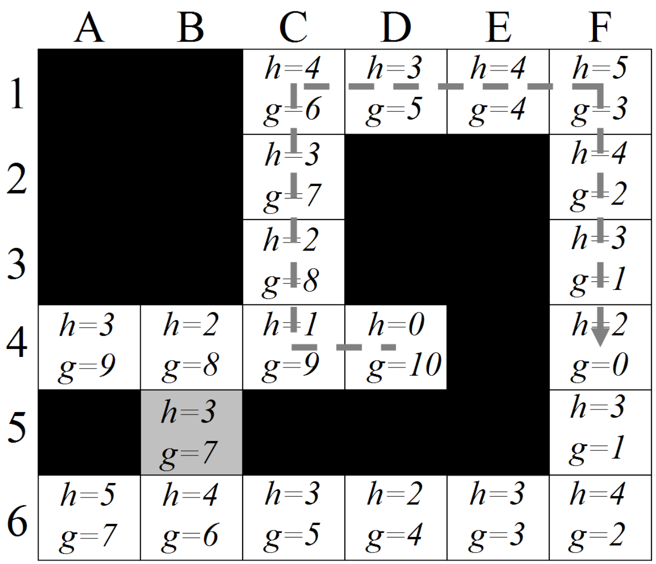
state with $v(X) < E_{X' \in succ(X,\pi(X))}\{c(X,\pi(X),X')+v(X')\}$

(state with negative Bellman error)

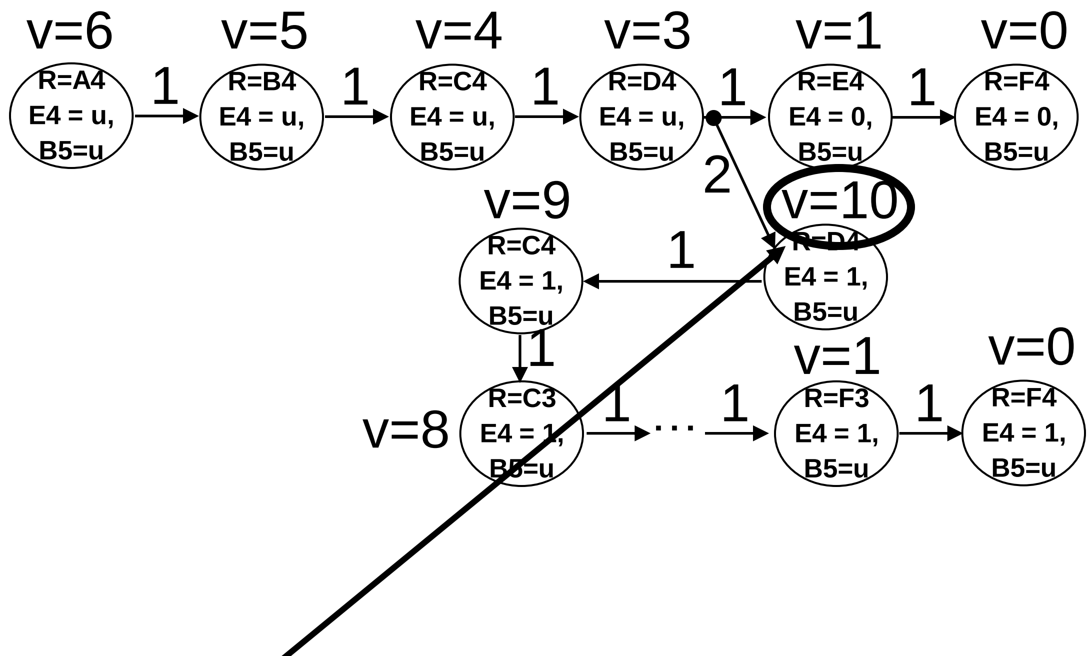
Run of PPCP

search backwards for a path from D4 to S_{goal}

assume: E4=1, B5=u



the policy after update:

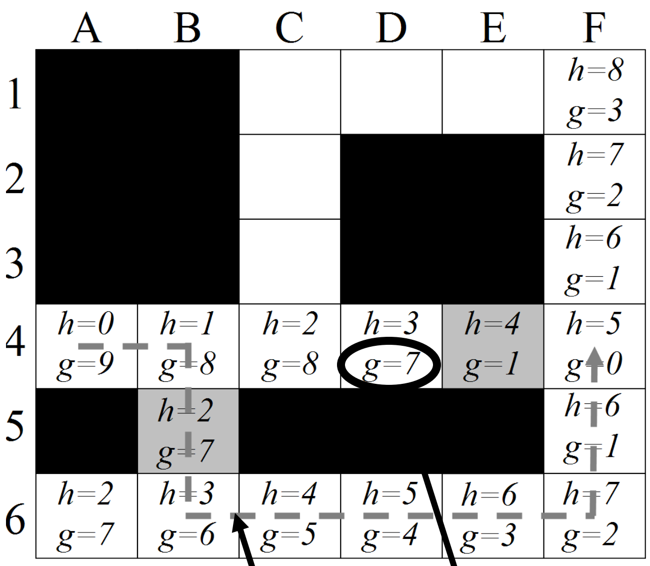


improved estimate

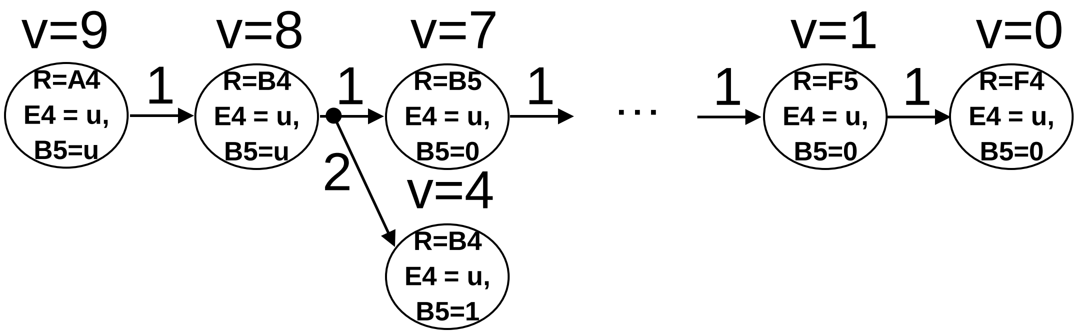
Run of PPCP

search backwards for a path from S_{start} to S_{goal}

assume: E4=u, B5=u



the policy after update:



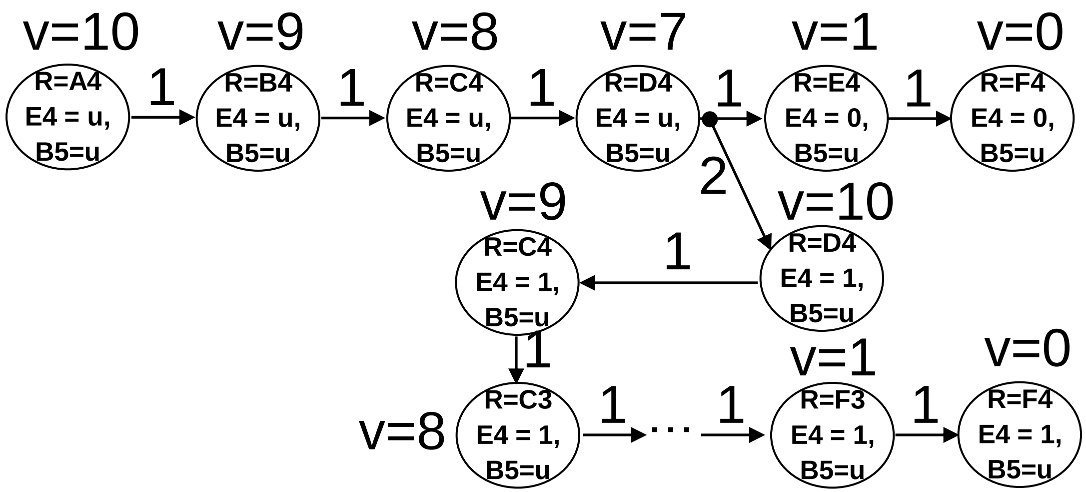
g(D4) = optimal expected cost-to-goal

0.5(1+1) + 0.5*(2+10)=7*

totally new path is found

Run of PPCP

the converged (optimal) policy
after 7 iterations



Theoretical properties:

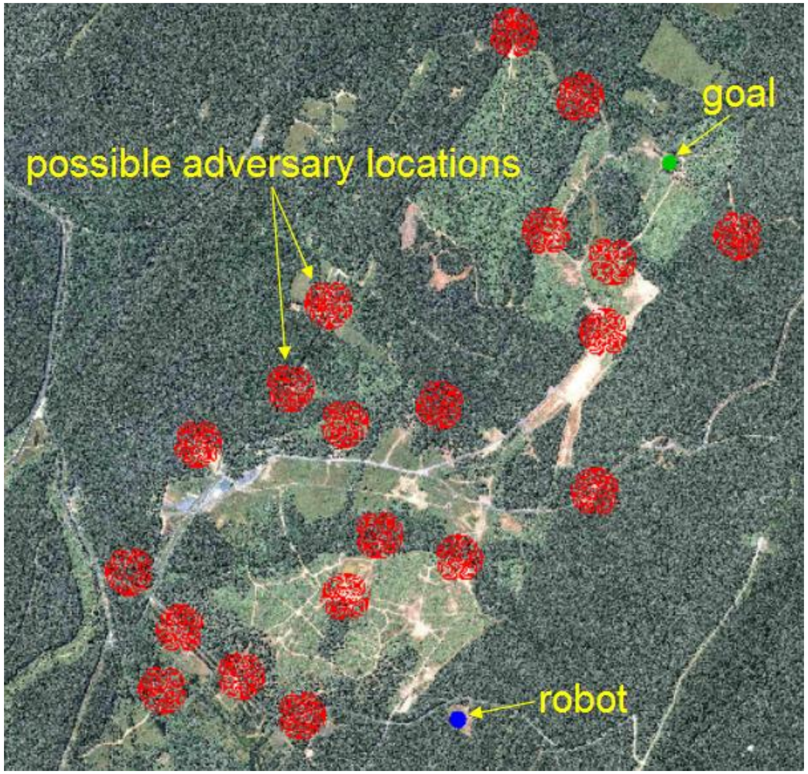
all states on the policy have $v(X) \geq E_{X' \in succ(X,\pi(X))}\{c(X,\pi(X),X')+v(X')\}$

the expected cost of the found policy is bounded from above by $v(X_{start})$

the found policy is guaranteed to be optimal if an optimal policy does not require remembering <u>*preferred*</u> *outcomes*

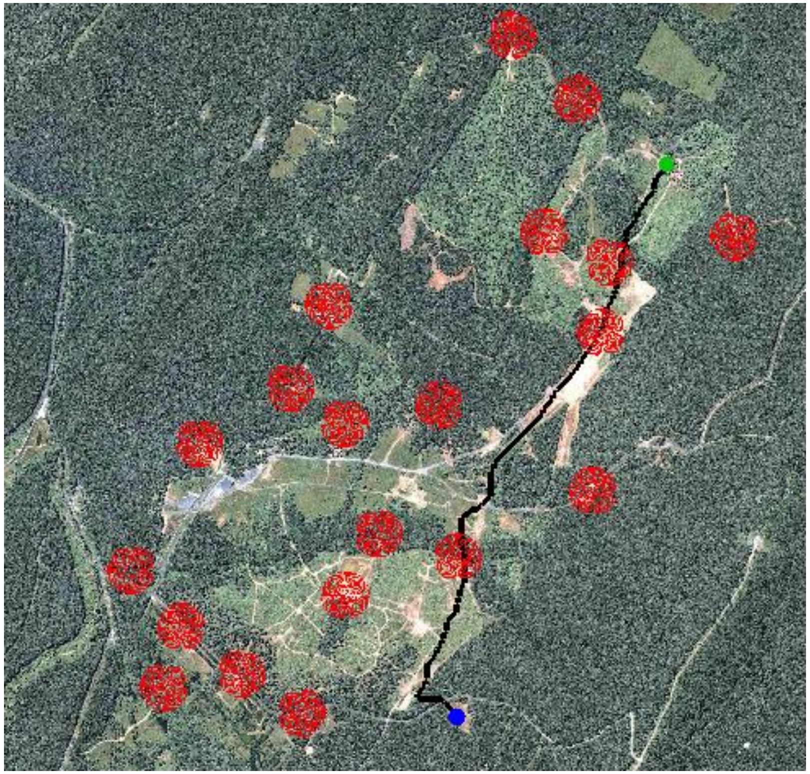
Solving Path Clearance using PPCP

environment size: 3.5km by 3.0km



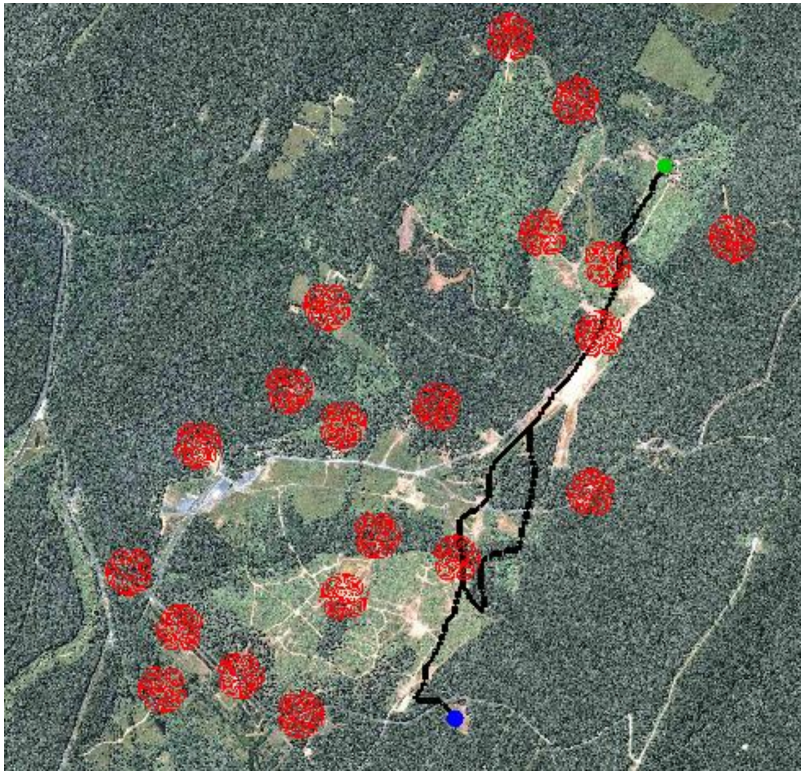
size of belief state-space: $500*500*3^{20}$

Solving Path Clearance using PPCP



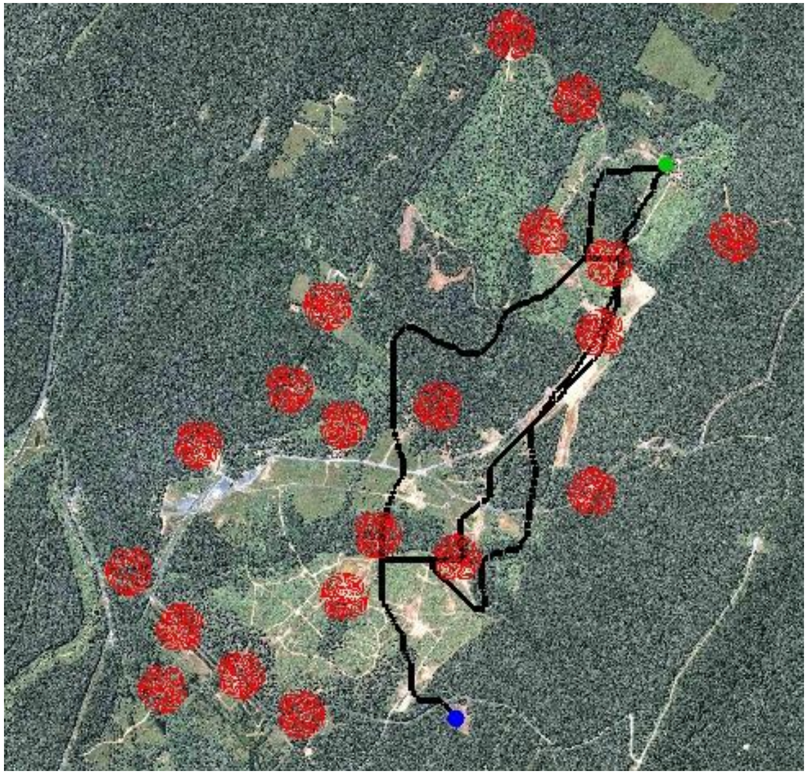
after first search (in few milliseconds)

Solving Path Clearance using PPCP



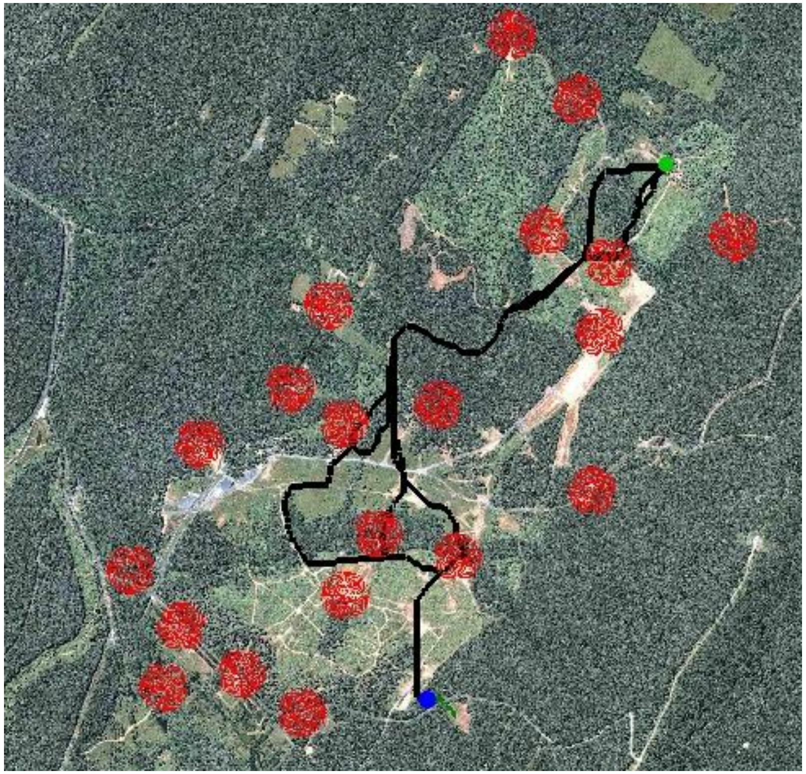
after second search (in few milliseconds)

Solving Path Clearance using PPCP



after 5 seconds

Solving Path Clearance using PPCP



after 30 seconds (converged)

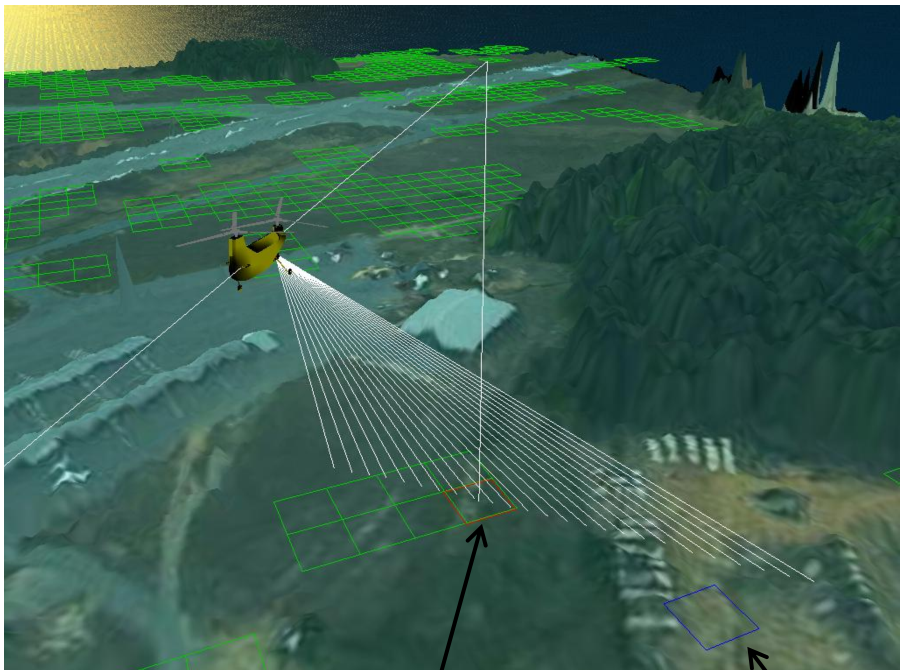
Solving Path Clearance using PPCP

Landing Site Selection using PPCP

Landing site selection problem: where to go + what to sense

- land safely **PREFER TO HAVE GOOD LANDING SITE**
- with minimum efforts
- as close to the desired goal as possible

policy produced by the planner



closest to the goal landing site of desired condifence

goal

Robot Navigation
in partially-known fractal environments

Maxim

size: 17 by 17

(the size of the belief state-space is up to *17*17*3*^{18})

# of unknowns	Percent Solved				Time to Convergence (in secs)				Solution Cost
	VI	LAO*	RTDP	PPCP	VI	LAO*	RTDP	PPCP	Same for All
6	92%	72%	100%	100%	7.7	43.9	0.4	0.1	112,284
10	—	36%	92%	100%	—	123.1	19.7	0.2	117,221
14	—	—	80%	100%	—	—	25.8	0.2	113,918
18	—	—	48%	100%	—	—	52.3	0.7	112,884

Interesting questions:

- need for memory about preferred outcomes when navigating random environments?

Robot Navigation in partially-known fractal environments

Maxim

size: 500 by 500

(the size of the belief state-space is up to $500*500*3^{25,000}$)

# of unknowns	Traversal Cost	
	PPCP	Freespace
1,000 (0.4%)	1,368,388	1,394,455
2,500 (1.0%)	1,824,853	1,865,935
5,000 (2.0%)	1,521,572	1,616,697
10,000 (4.0%)	1,626,413	1,685,717
25,000 (10.0%)	1,393,694	1,484,018

Interesting questions: freespace assumption vs. probabilistic plan.

- benefits of probabilistic planning are consistent but not high
- on the other hand, using PPCP for path clearance can save over 35% in execution cost

Table of Contents

- Modeling Planning Domains
  - Graphs, MDPs
- Planning Problems and Strategies
  - Localization, Mapping, Navigation in Unknown Terrain
  - Agent-Centered Search, Assumptive Planning
- Efficient Implementations of Planning Strategies
  - Incremental Heuristic Search

15 Minute Break

  - Real-Time Heuristic Search
  - Planning with Preferences on Uncertainty
  - Planning with Varying Abstractions

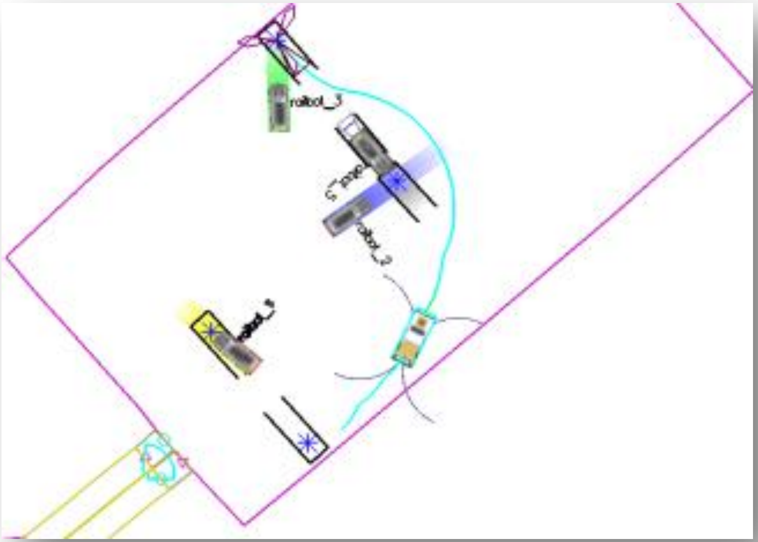
Case Study:
Planning in Dynamic Environments

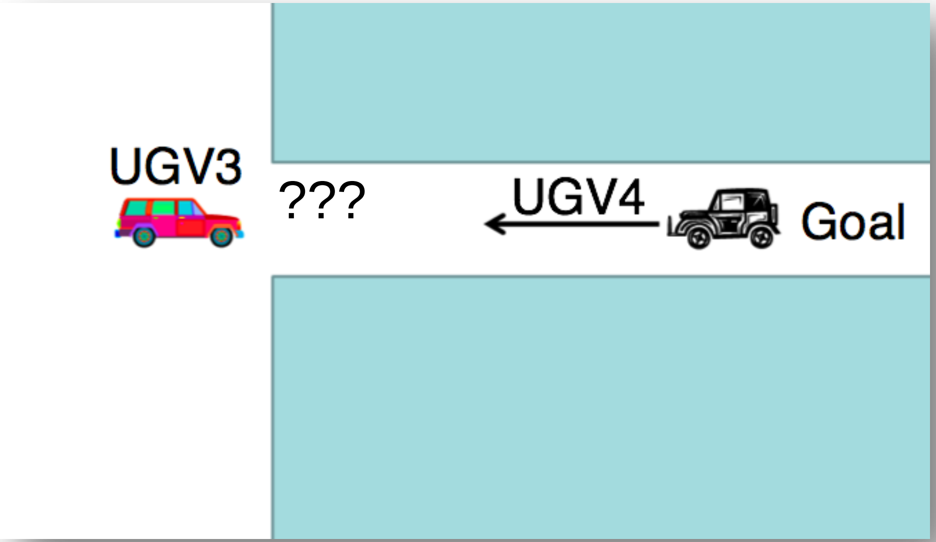
- Shows the actual application of some of the presented techniques

Robust goal-directed behavior in Dynamic Environments

Most Real-time Approaches

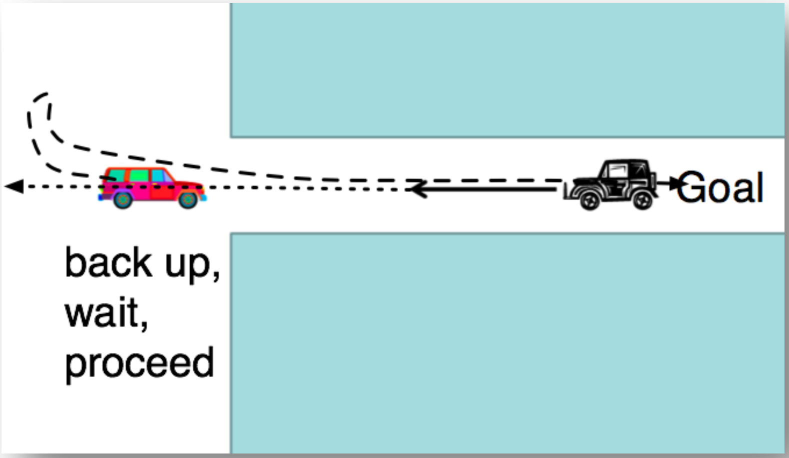
- Project the dynamic obstacles onto the static 2D map by assigning high cost to cells that lie on the obstacles' expected paths
  - Fast but can be highly suboptimal
  - Can cause the robot to get stuck





Optimal Approaches

- Produce high dimensional time-parameterized trajectories all the way to the goal (i.e. <x, y, ϑ, … , t>) [Fiorini & Shiller, ’98; Fujimura & Samet, ‘93; van den Berg & Overmars, ’06]
  - Should take into account vehicle dynamics
  - Computationally expensive and slow
  - By the time planning is finished, the situation, with respect to dynamic obstacles, may change



Key Idea in Time-bounded Lattice

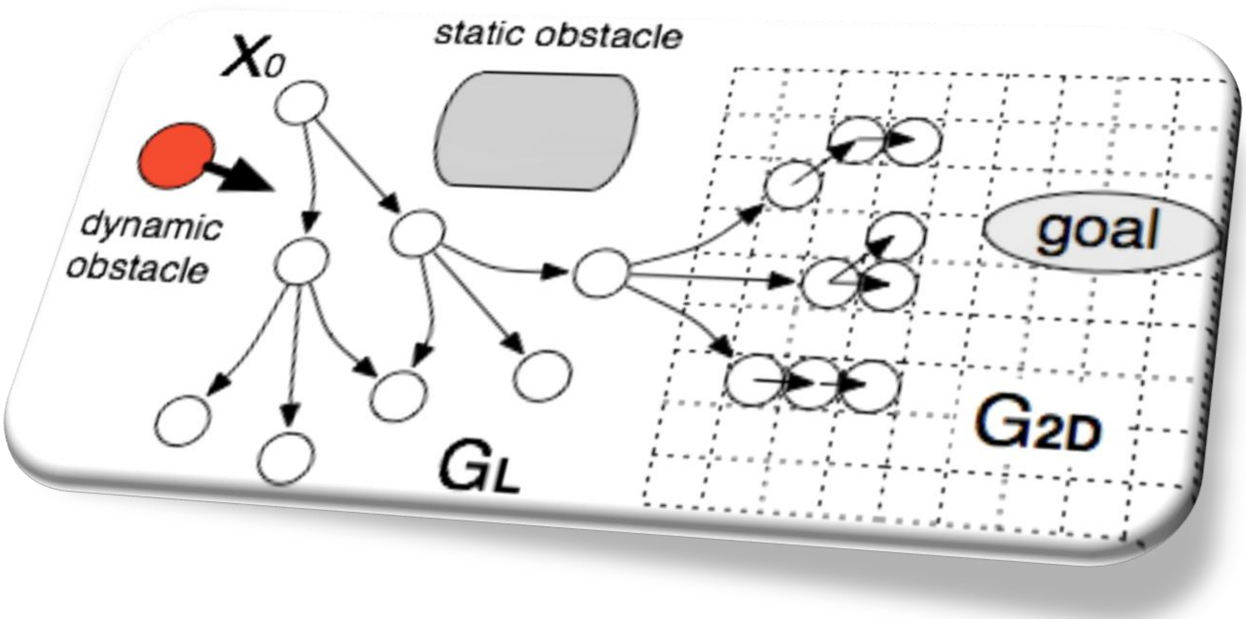
- Main Observations:

  - The uncertainty in the obstacle motion prediction is usually quite high, so planning over time far into the future does not make sense.
    - Uncertainty in past observations
    - Uncertainty in future trajectories

  - The robot will be able to re-plan avoidance maneuvers as it gets closer

Key Idea in Time-bounded Lattice

- Combine planning dynamically feasible time-parameterized trajectories with low-dimensional planning w/o time



- Automatically reason about the extent of planning in time based on uncertainty in future obstacle trajectories

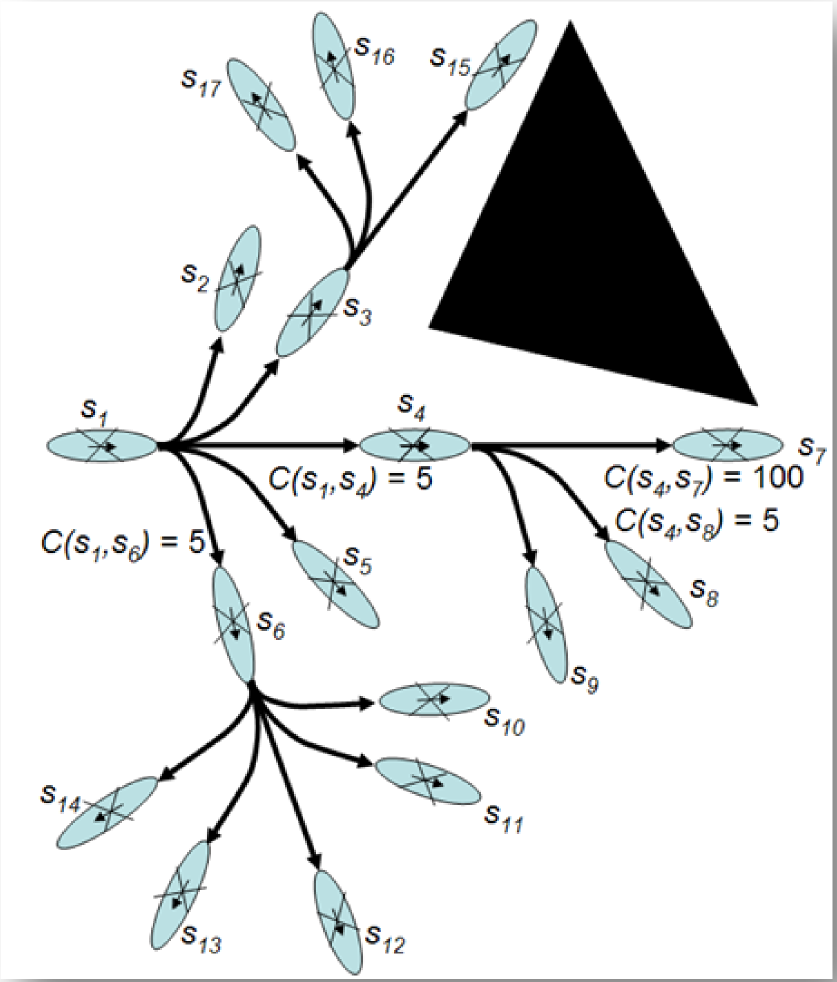
Key Idea in Time-bounded Lattice

- Combine planning dynamically feasible time-parameterized trajectories with low-dimensional planning w/o time
  - high-dimensional agent-centered search combined with low-dimensional planning with freespace assumption
  - freespace assumption refers to assuming “no dynamic obstacles”

- Automatically reason about the extent of planning in time based on uncertainty in future obstacle trajectories

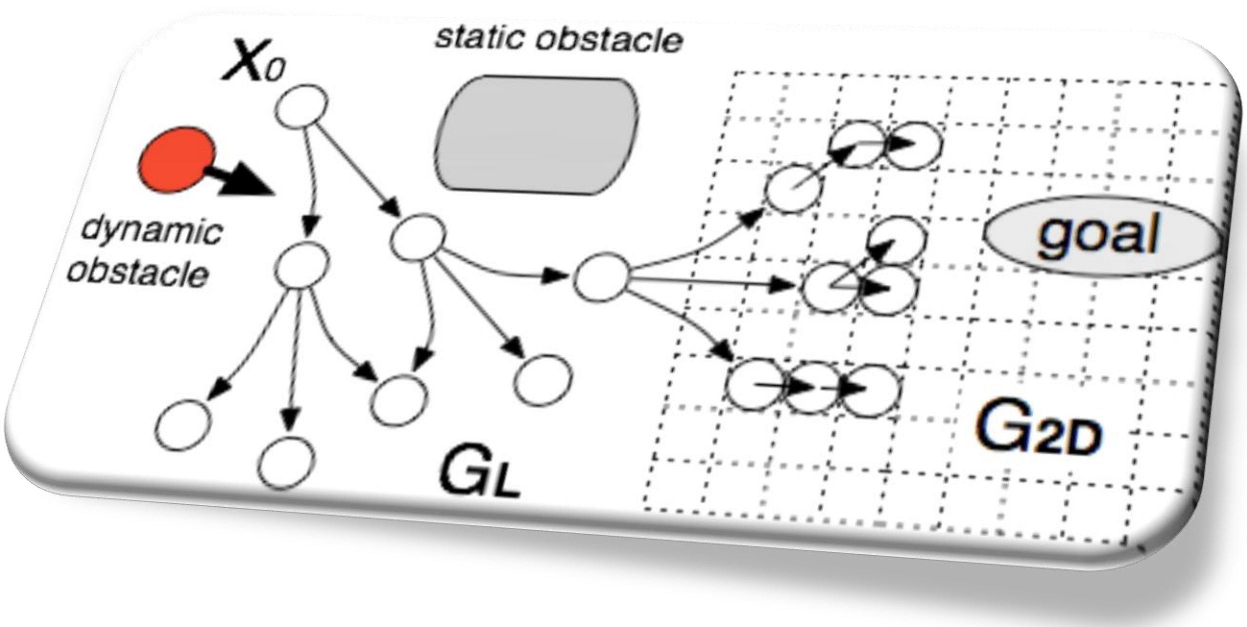
Lattice Graph

- Lattice graph construction [Pivtoraiko & Kelly, '05]:
  - Uses dynamically-feasible motion primitives to produce successors
  - Motion primitives can be generated for a particular robot platform
  - Transition costs can assigned to successors based on length, heading change, etc
  - States that collide with obstacles receive high costs and/or can be discarded (not shown)



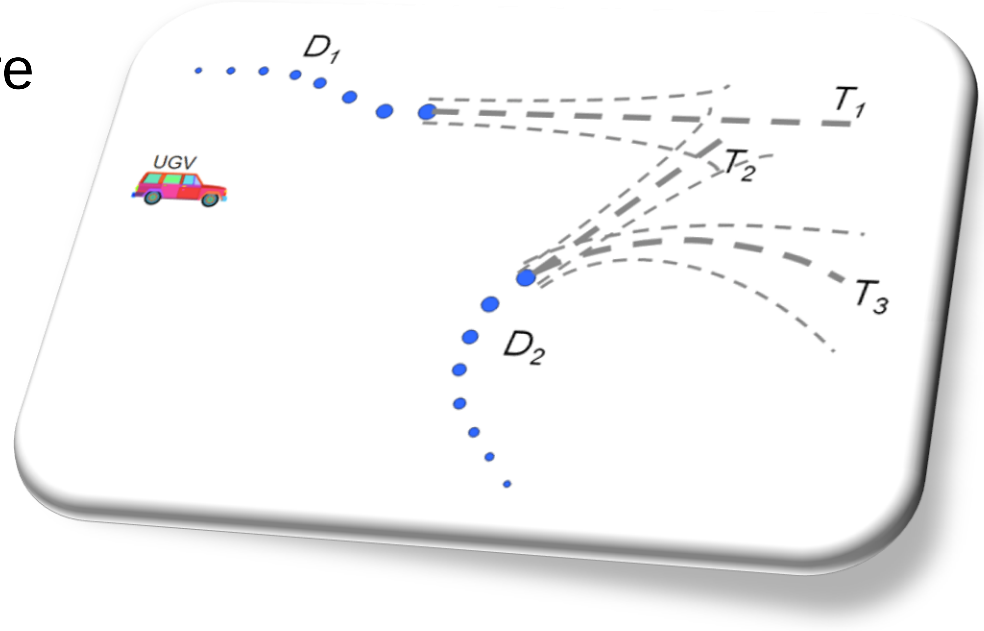
Time-bounded Lattice

- Start planning with time in high dimensional lattice ($<x, y, \vartheta, v, \omega, t>$)
- Determine when it is safe to ignore the obstacles based on their estimated future position uncertainty (find T_{max})
- All states with $t > T_{max}$ are projected onto a graph w/o time (i.e., 2D grid)
- ARA* is used to construct and search the graph



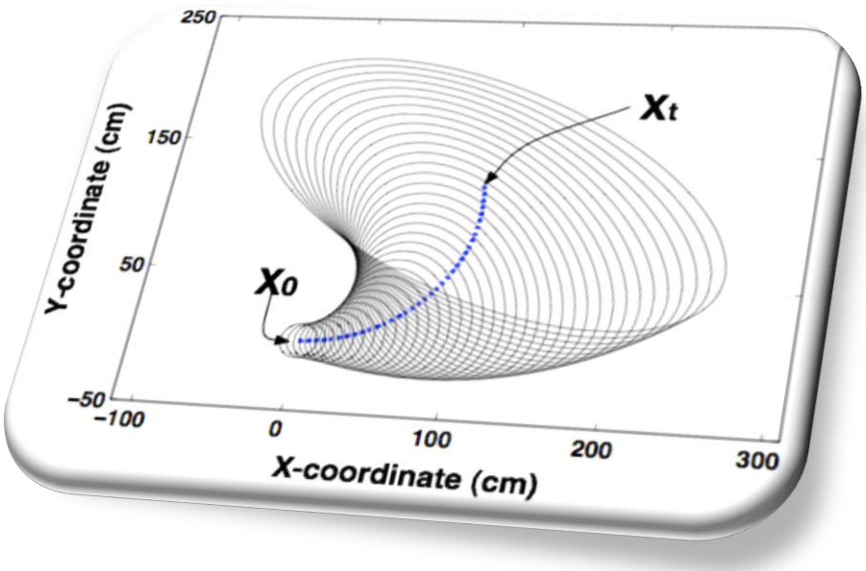
Obstacle Representation

- Time-parameterized pose distribution
- Expected poses of obstacles can be extrapolated into the future given past observations and their motion models
- Multimodal hypotheses are supported (i.e. T_2, T_3)



Obstacle Representation

- 3D Gaussian was chosen to represent the pose uncertainty of the dynamic obstacles
  - < x, y, ϑ > (3x3 cov. matrix)
  - Differential drive motion model
  - EKF prediction step

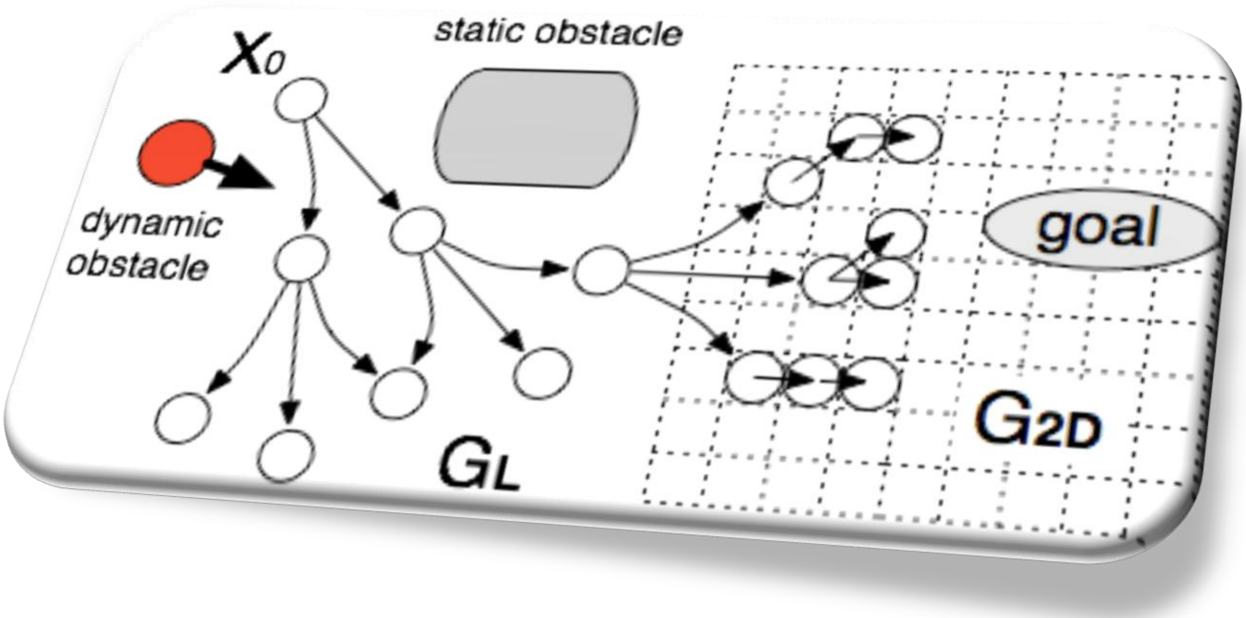


- Planner is not restricted to any particular obstacle uncertainty model

Estimating Collision Cost

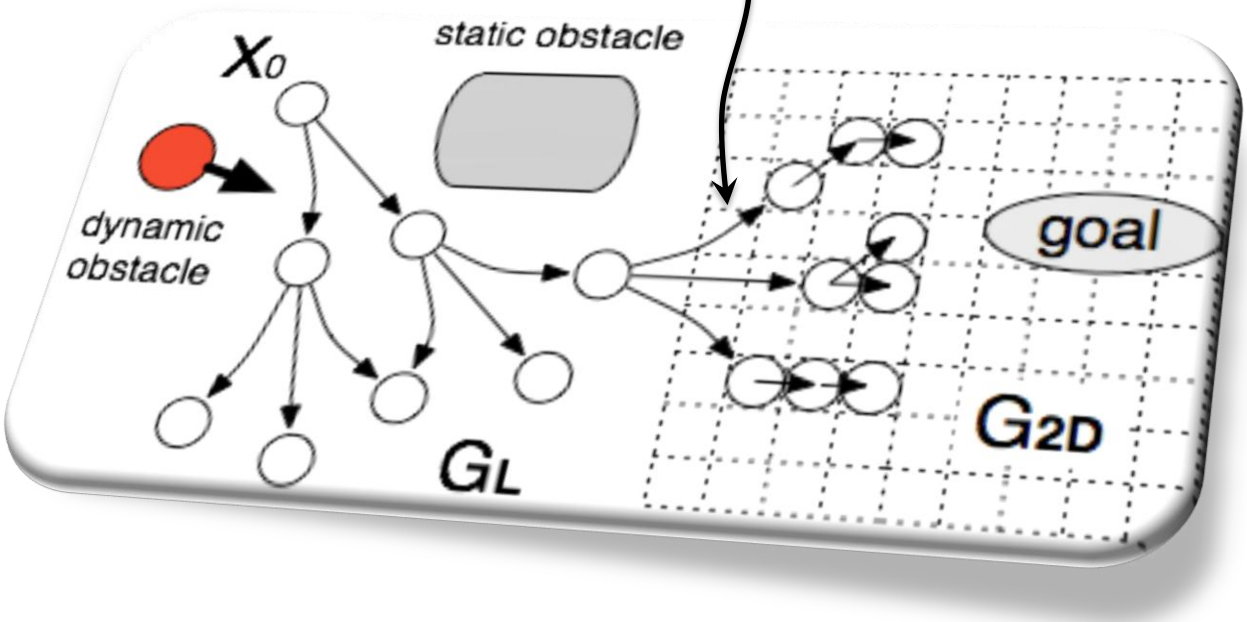
- For every action, we can now compute the probability of colliding with a dynamic obstacle

- The cost of the state transition is proportional to the probability of collision



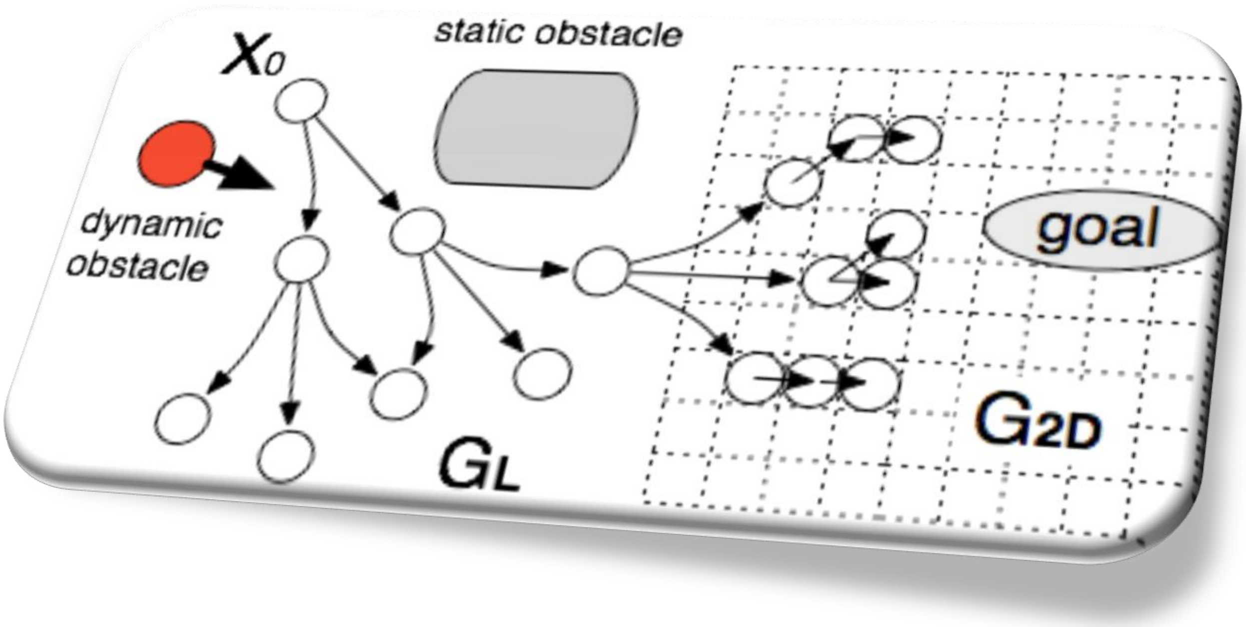
Computing T_{max}

- Probability of collision at time *t* is upper-bounded by P_{max} - the integral over the robot footprint at the mean of the distribution at time *t*
- T_{max} is time *t* when P_{max} is negligible



Collision Cost for 2D Grid

- Only take into account static obstacles



Advantages of Time-bounded Lattice

- Output of the planner can be fed directly into vehicle controls
- Simple low-d planning if dynamic obstacles are absent
- Full 6D trajectories if obstacle motion prediction is accurate
- Automatically balances between the two extremes

Example of Planning with Time-bounded Lattice

Summary

- Planning with freespace assumption and its anytime/incremental implementations
- Agent-centered search and its incremental implementations
- Probabilistic planning with preferences on uncertainty

  - each strategy results in “good” run-time behavior in some domains
  - but may result in highly suboptimal run-time behavior in other domains
  - in some domains may also be beneficial to combine the strategies

Summary

- Solving complex planning problems by running a series of A*-like searches (ARA*, PPCP, R*, MCP,…)
  - typically easily to implement
  - makes use of heuristics
  - automatically focusses on relevant states
  - provides theoretical guarantees
  - general
  - often provides anytime behavior

Concluding Remarks

- Joint work with
  - S. Chitta, B. Cohen, K. Daniel, A . Felner, D. Ferguson, G. Gordon, S. Greenberg, W. Halliburton, A. Kushleyev, A. Mudgal, A. Nash, A. Ranganathan, Y. Smirnov, A. Stentz, X. Sun, S. Thrun and C. Tovey
- Funded in part by
  - NSF, DARPA, ARL, ONR, Willow Garage, IBM and JPL
- For more information see
  - idm-lab.org/projects.html
  - www.seas.upenn.edu/~maximl
- Download software from
  - idm-lab.org/project-a.html
  - www.seas.upenn.edu/~maximl/software.html (SBPL library)
  - SBPL and SBPL-based motion planners are also available as part of ROS packages (http://www.ros.org/wiki/sbpl)