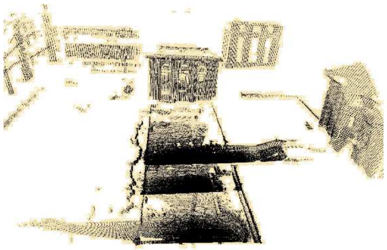
Planning in Dynamic Environments

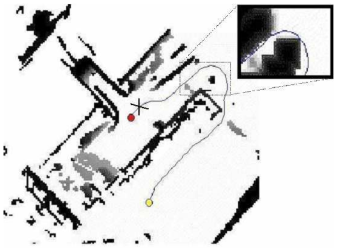
Maxim Likhachev
Robotics Institute
Carnegie Mellon University

Autonomous Agents in Dynamic Environments

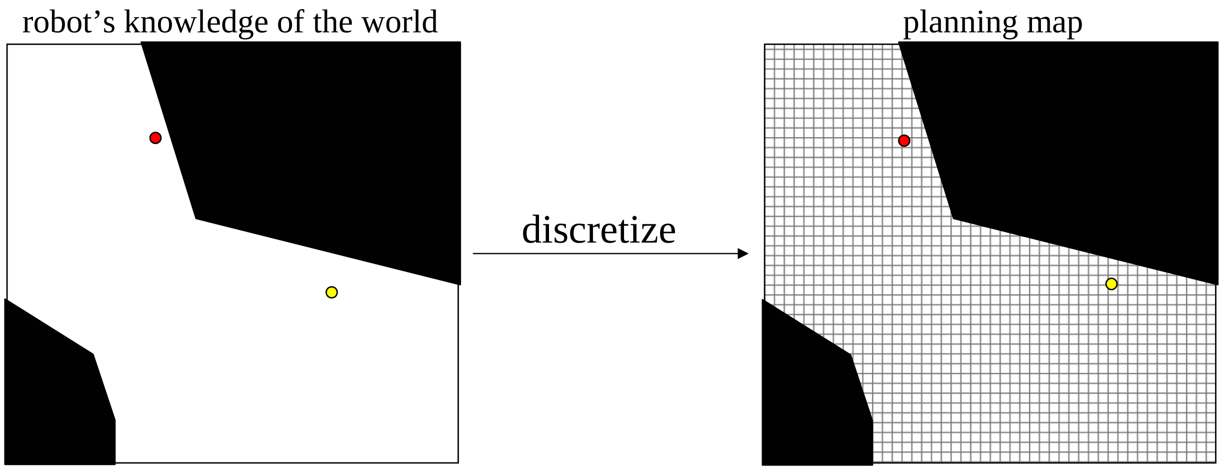
ATRV robot

segbot robot

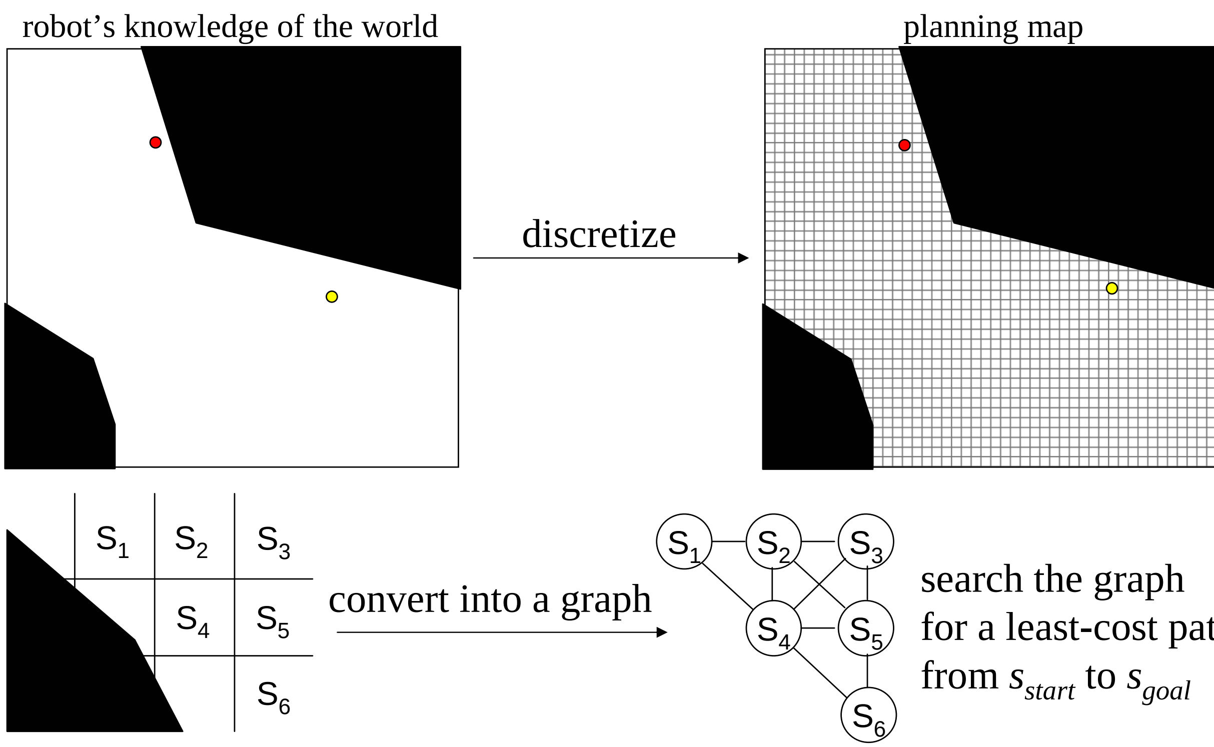

3D map


2D map

Search-based Planning



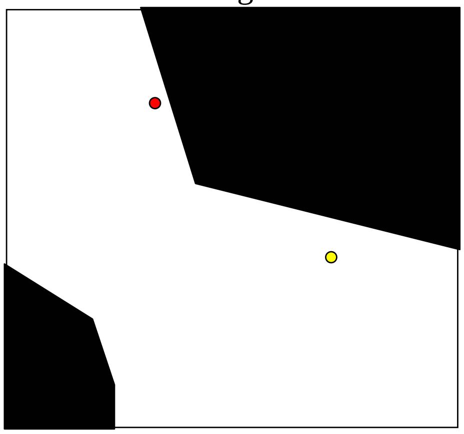
Search-based Planning



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

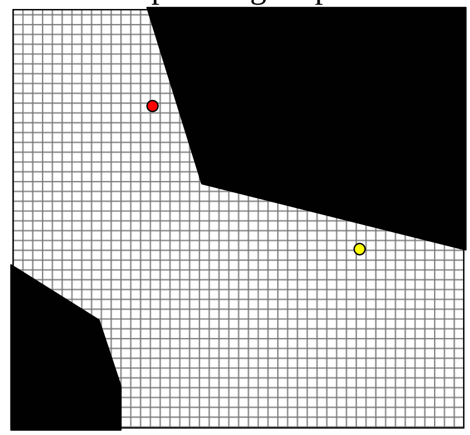
Search-based Planning

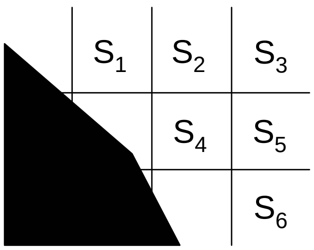
robot's knowledge of the world



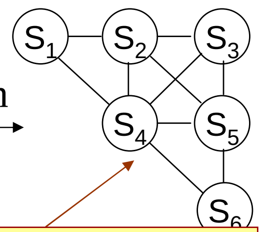
discretize

planning map





convert into a graph

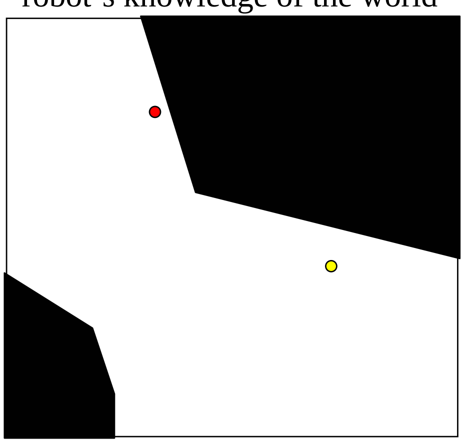


eight-connected grid

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

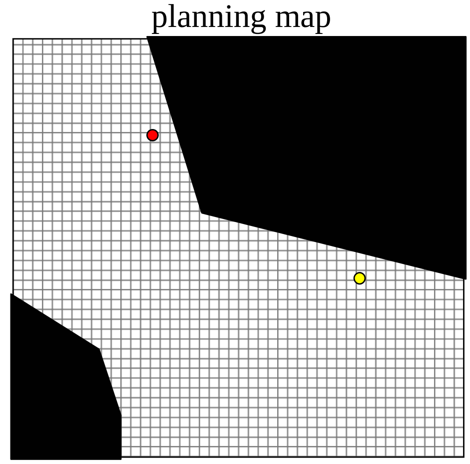
Search-based Planning

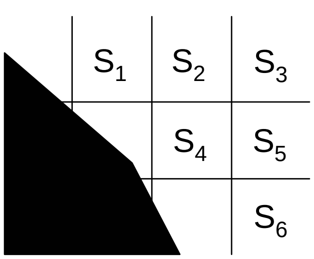
robot's knowledge of the world



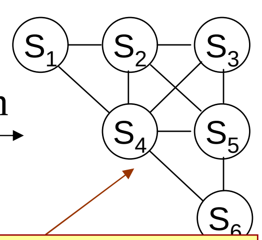
discretize

planning map





convert into a graph

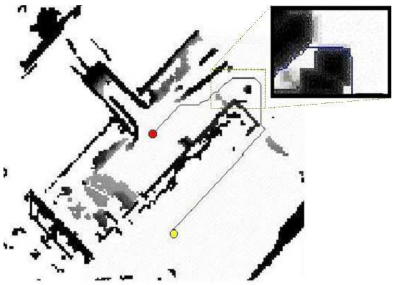


eight-connected grid

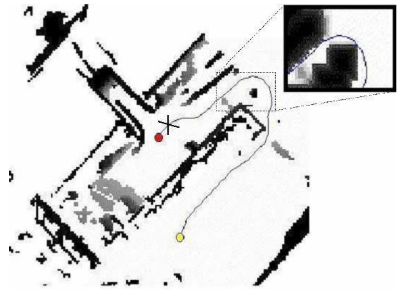
search the graph for a least-cost path from s_{start} to s_{goal}

why?

High Dimensional Search-based Planning

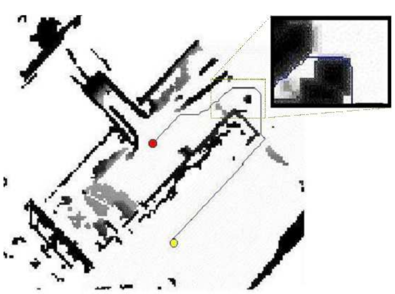


2D (x, y) planning
54K states



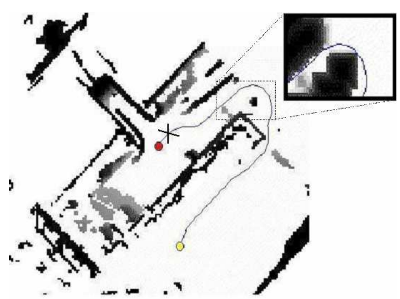
4D (x, y, Θ, V) planning
over 20 million states

High Dimensional Search-based Planning



2D (x, y) planning
54K states

fast planning
slow execution

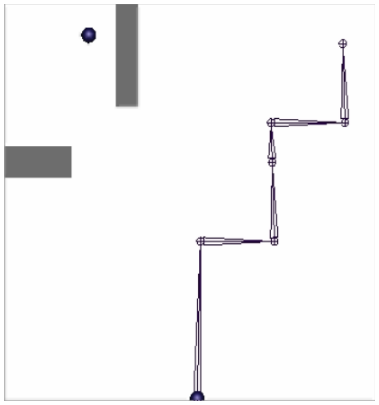


4D (x, y, Θ, V) planning
over 20 million states

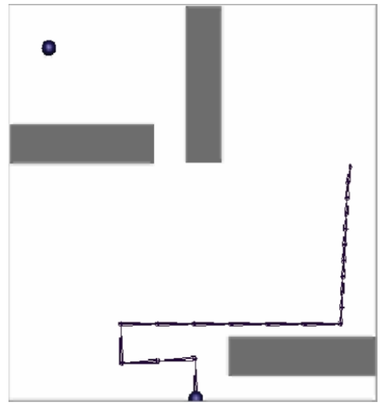
slow planning
fast execution

why?

High Dimensional Search-based Planning



6 DOF robot arm
$> 3*10^9$ states



20 DOF robot arm
$> 10^{26}$ states

Planning in Real World

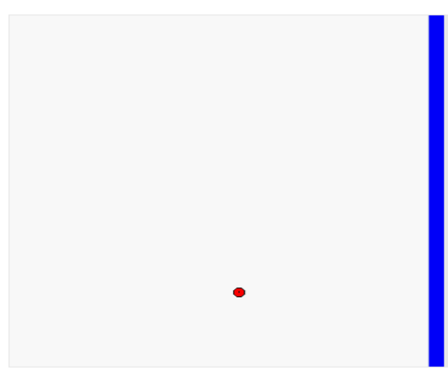
- need to re-plan often due to
  - changes in the environment
    - navigation with people around
    - autonomous car driving with other cars on the road
  - inaccuracy in the model of the environment
  - errors in the position estimate

Planning in Real World

- need to re-plan often due to
  - changes in the environment
    - navigation with people around
    - autonomous car driving with other cars on the road
  - inaccuracy in the model of the environment
  - errors in the position estimate
- **need to re-plan fast!**

Planning in Real World

- need to re-plan often due to
  - changes in the environment
    - navigation with people around
    - autonomous car driving with other cars on the road
  - inaccuracy in the model of the environment
  - errors in the position estimate
- **need to re-plan fast!**



4D planning with Anytime D (Anytime Dynamic A*)*

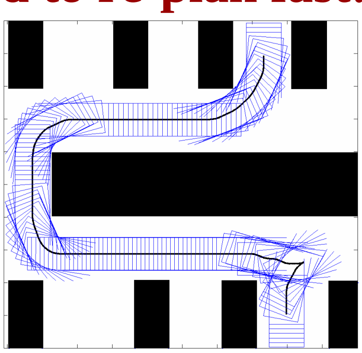
Planning in Real World

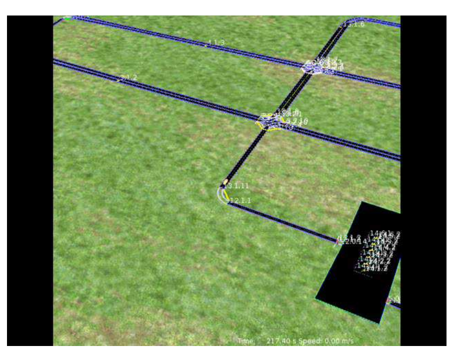
- need to re-plan often due to
  - changes in the environment
    - navigation with people around
    - autonomous car driving with other cars on the road
  - inaccuracy in the model of the environment
  - errors in the position estimate
- **need to re-plan fast!**

4D planning with Anytime D (Anytime Dynamic A*)*

Planning in Real World

- need to re-plan often due to
  - changes in the environment
    - navigation with people around
    - autonomous car driving with other cars on the road
  - inaccuracy in the model of the environment
  - errors in the position estimate
- **need to re-plan fast!**





3D parking planning with Anytime D (Anytime Dynamic A*) for the next DARPA Grand Challenge*

Planning in Real World

- Anytime planning algorithms (e.g., ARA*- anytime version of A*)
  - find first possibly highly-suboptimal solution quickly, use the remaining time to improve it
  - allow to meet time constraints

- Replanning algorithms (e.g., D* and D* Lite – incremental versions of A*)
  - speed up the task of repeated planning by reusing previous planning efforts
  - well-suited for dynamic and/or partially known environments

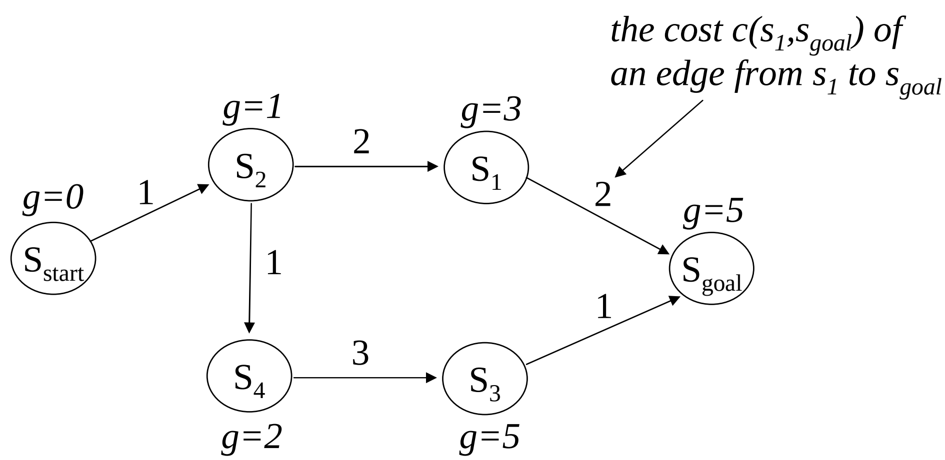
- Anytime replanning algorithms (e.g., Anytime D* - anytime incremental A*)
  - combine the benefits of the two

Search for a Least-cost Path

- Compute optimal g-values for relevant states

  - $g(s)$ – an estimate of the cost of a least-cost path from s_{start} to s

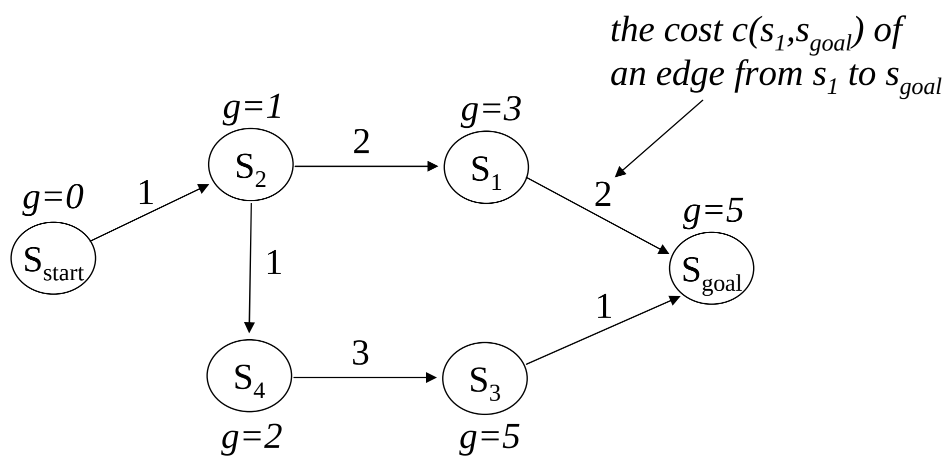
  - optimal values satisfy: $g(s) = \min_{s'' \in pred(s)} g(s'') + c(s'',s)$



Search for a Least-cost Path

- Compute optimal g-values for relevant states
  - *g(s)* – an estimate of the cost of a least-cost path from s_{start} to s
  - optimal values satisfy: $g(s) = \min_{s'' \in pred(s)} g(s'') + c(s'',s)$

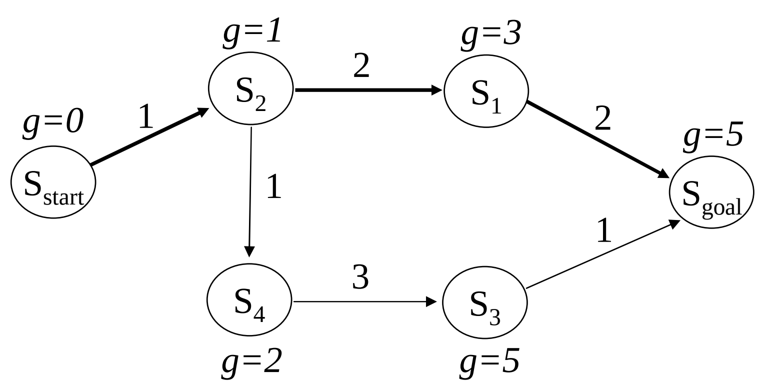
why?



Search for a Least-cost Path

- Least-cost path is a greedy path computed by backtracking:
  - start with s_{goal} and from any state *s* move to the predecessor state *s'* such that

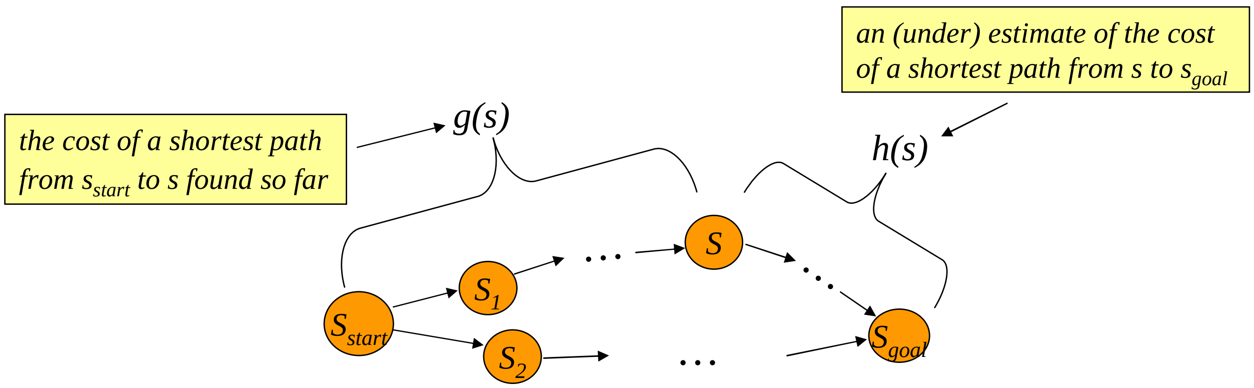
$$s' = \arg\min_{s'' \in pred(s)} (g(s'') + c(s'', s))$$



A* Search

- Computes optimal g-values for relevant states

- At any point of time:



A* Search

- Computes optimal g-values for relevant states

Main function

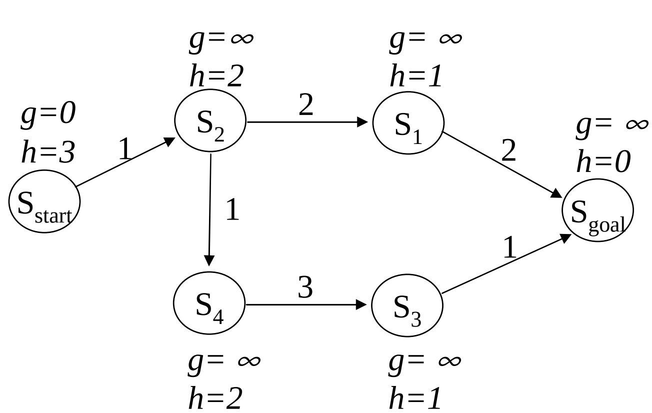
g(s_{start}) = 0; all other *g*-values are infinite*; OPEN = {s_{start}};*
ComputePath();
publish solution;

set of candidates for expansion

ComputePath function

while(s_{goal} is not expanded)
  remove *s* with the smallest *[f(s) = g(s)+h(s)]* from *OPEN*;
  expand *s*;

for every expanded state
g(s) is optimal
(if heuristics are consistent)



A* Search

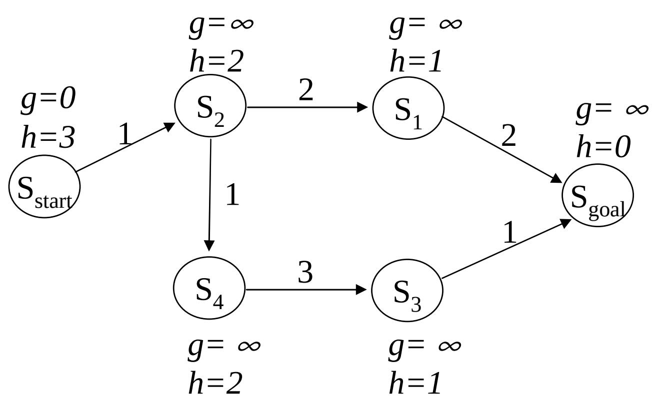
- Computes optimal g-values for relevant states

ComputePath function

while(s_{goal} is not expanded)

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

  expand s;



A* Search

- Computes optimal g-values for relevant states

ComputePath function

while(s_{goal} is not expanded)

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

  insert s into *CLOSED*;

  for every successor s' of s such that s' not in *CLOSED*

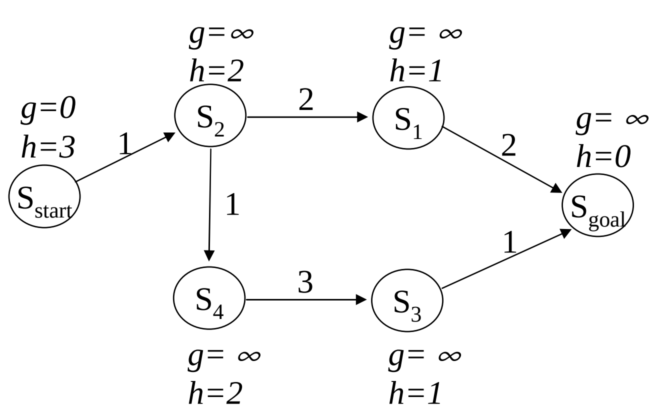
    if $g(s') > g(s) + c(s,s')$

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

      insert s' into *OPEN*;

set of states that have already been expanded

tries to decrease $g(s')$ using the found path from s_{start} to s



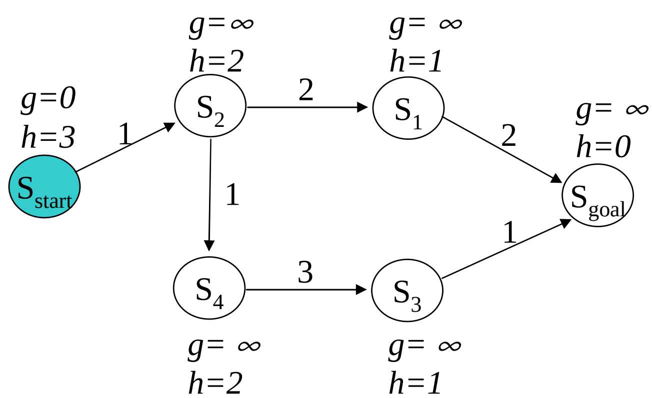
A* Search: Example

- Computes optimal g-values for relevant states

ComputePath function

while(s_{goal} is not expanded)
  remove s with the smallest $[f(s) = g(s)+h(s)]$ from *OPEN*;
  insert s into *CLOSED*;
  for every successor s' of s such that s' not in *CLOSED*
    if $g(s') > g(s) + c(s,s')$
      $g(s') = g(s) + c(s,s')$;
      insert s' into *OPEN*;

CLOSED = {}
OPEN = $\{s_{start}\}$
next state to expand: s_{start}



A* Search: Example

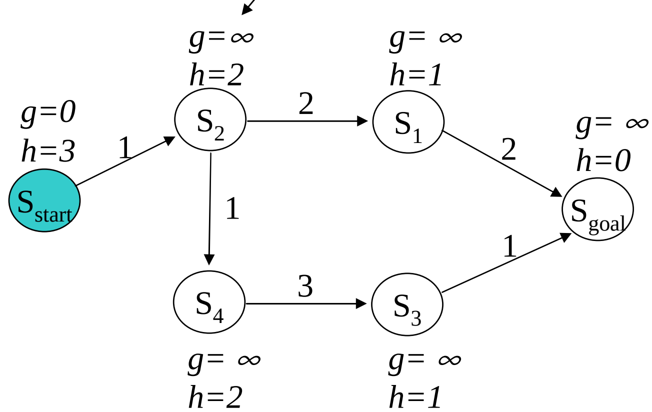
- Computes optimal g-values for relevant states

ComputePath function

while(s_{goal} is not expanded)
  remove s with the smallest $[f(s) = g(s)+h(s)]$ from *OPEN*;
  insert s into *CLOSED*;
  for every successor s' of s such that s' not in *CLOSED*
    if $g(s') > g(s) + c(s,s')$
      $g(s') = g(s) + c(s,s')$;
      insert s' into *OPEN*;

$g(s_2) > g(s_{start}) + c(s_{start},s_2)$

CLOSED = {}
OPEN = $\{s_{start}\}$
next state to expand: s_{start}



A* Search: Example

- Computes optimal g-values for relevant states

ComputePath function

while(s_{goal} is not expanded)

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

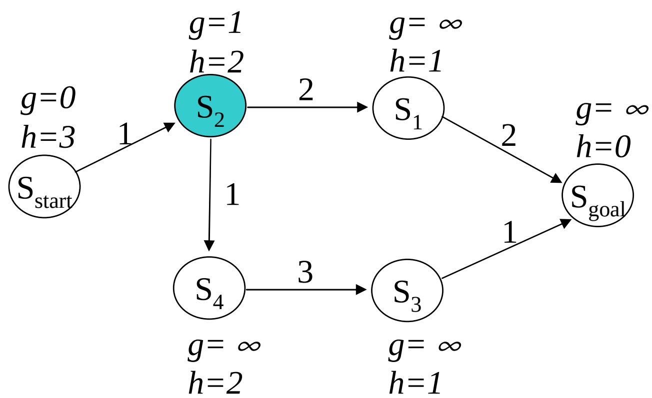
insert *s* into *CLOSED*;

for every successor *s'* of *s* such that *s'* not in *CLOSED*

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

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

insert *s'* into *OPEN*;



A* Search: Example

- Computes optimal g-values for relevant states

ComputePath function

while(s_{goal} is not expanded)

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

insert *s* into *CLOSED*;

for every successor *s'* of *s* such that *s'* not in *CLOSED*

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

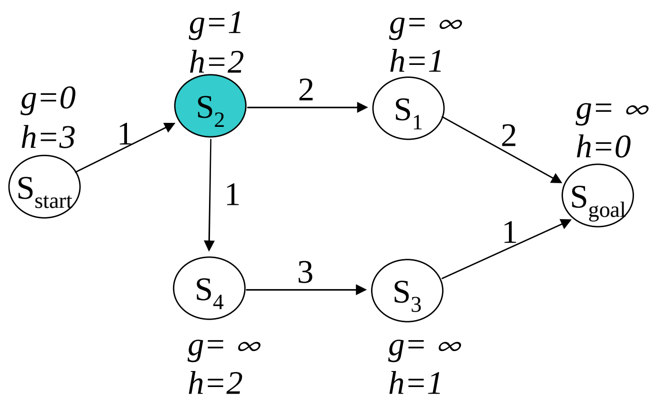
$g(s') = g(s) + c(s,s')$;

insert *s'* into *OPEN*;

$CLOSED = \{s_{start}\}$

$OPEN = \{s_2\}$

next state to expand: s_2



A* Search

- Computes optimal g-values for relevant states

ComputePath function

while(s_{goal} is not expanded)

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

  insert s into *CLOSED*;

  for every successor s' of s such that s' not in *CLOSED*

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

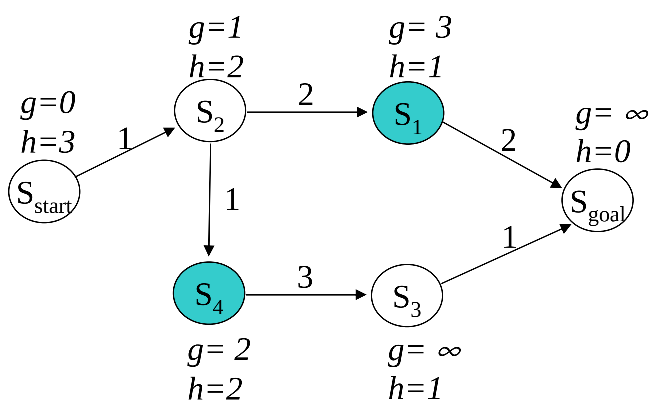
      $g(s') = g(s) + c(s,s');$

      insert s' into *OPEN*;

$CLOSED = \{s_{start}, s_2\}$

$OPEN = \{s_1, s_4\}$

next state to expand: s_1



A* Search

- Computes optimal g-values for relevant states

ComputePath function

while(s_{goal} is not expanded)

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

  insert s into *CLOSED*;

  for every successor s' of s such that s' not in *CLOSED*

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

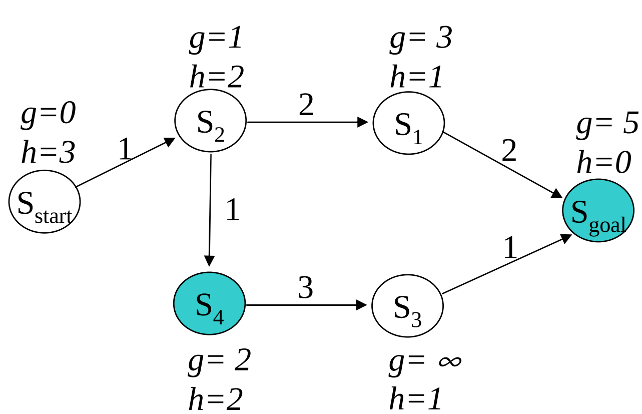
      $g(s') = g(s) + c(s,s');$

      insert s' into *OPEN*;

$CLOSED = \{s_{start}, s_2, s_1\}$

$OPEN = \{s_4, s_{goal}\}$

next state to expand: s_4



A* Search

- Computes optimal g-values for relevant states

ComputePath function

while(s_{goal} is not expanded)

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

insert *s* into *CLOSED*;

for every successor *s'* of *s* such that *s'* not in *CLOSED*

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

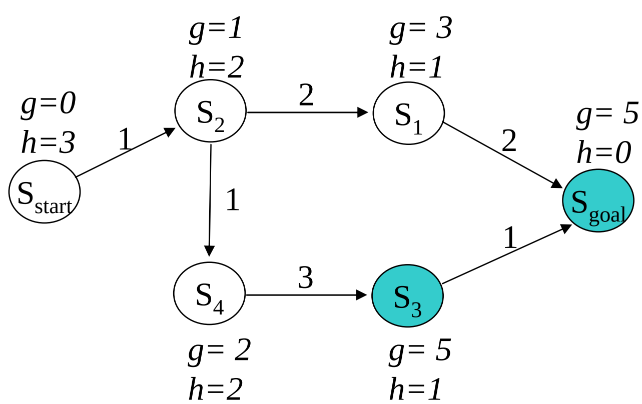
$g(s') = g(s) + c(s,s')$;

insert *s'* into *OPEN*;

$CLOSED = \{s_{start}, s_2, s_1, s_4\}$

$OPEN = \{s_3, s_{goal}\}$

next state to expand: s_{goal}



A* Search

- Computes optimal g-values for relevant states

ComputePath function

while(s_{goal} is not expanded)

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

insert *s* into *CLOSED*;

for every successor *s'* of *s* such that *s'* not in *CLOSED*

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

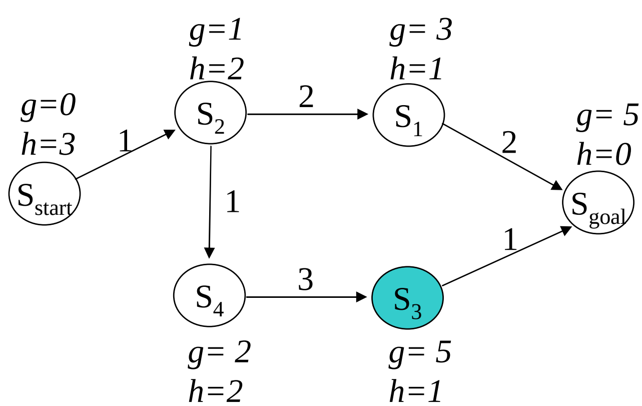
$g(s') = g(s) + c(s,s')$;

insert *s'* into *OPEN*;

$CLOSED = \{s_{start}, s_2, s_1, s_4, s_{goal}\}$

$OPEN = \{s_3\}$

done

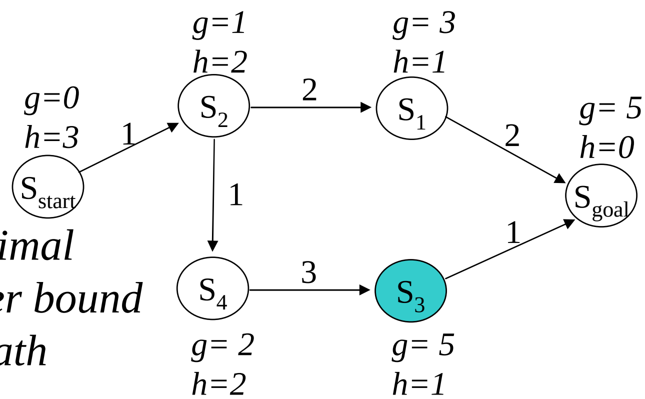


A* Search

- Computes optimal g-values for relevant states

ComputePath function

while(s_{goal} is not expanded)
  remove s with the smallest $[f(s) = g(s)+h(s)]$ from *OPEN*;
  insert s into *CLOSED*;
  for every successor s' of s such that s' not in *CLOSED*
    if $g(s') > g(s) + c(s,s')$
      $g(s') = g(s) + c(s,s');$
      insert s' into *OPEN*;



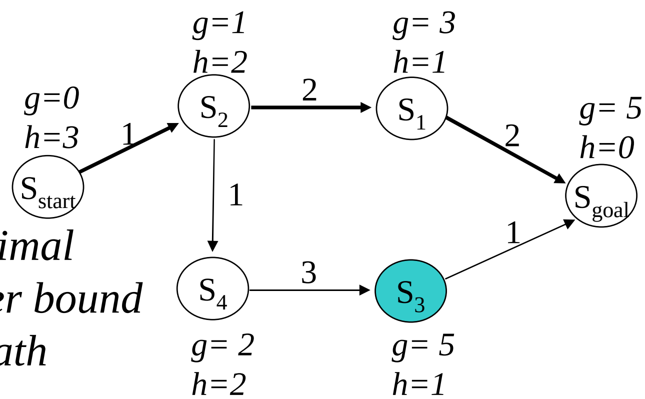
for every expanded state g(s) is optimal
for every other state g(s) is an upper bound
we can now compute a least-cost path

A* Search

- Computes optimal g-values for relevant states

ComputePath function

while(s_{goal} is not expanded)
  remove s with the smallest $[f(s) = g(s)+h(s)]$ from *OPEN*;
  insert s into *CLOSED*;
  for every successor s' of s such that s' not in *CLOSED*
    if $g(s') > g(s) + c(s,s')$
      $g(s') = g(s) + c(s,s');$
      insert s' into *OPEN*;



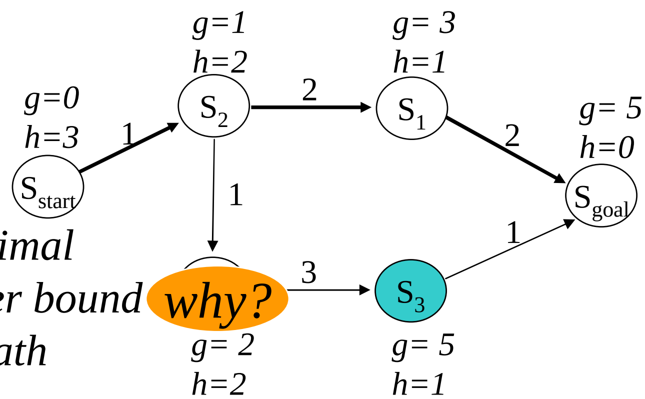
for every expanded state g(s) is optimal
for every other state g(s) is an upper bound
we can now compute a least-cost path

A* Search

- Computes optimal g-values for relevant states

ComputePath function

while(s_{goal} is not expanded)
  remove s with the smallest $[f(s) = g(s)+h(s)]$ from *OPEN*;
  insert s into *CLOSED*;
  for every successor s' of s such that s' not in *CLOSED*
    if $g(s') > g(s) + c(s,s')$
      $g(s') = g(s) + c(s,s')$;
      insert s' into *OPEN*;



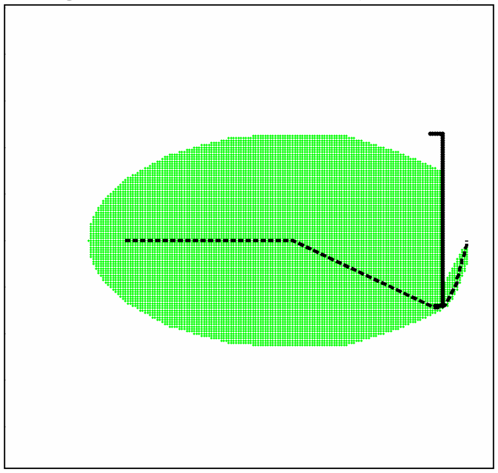
for every expanded state g(s) is optimal
for every other state g(s) is an upper bound
we can now compute a least-cost path

Weighted A*

- Expands states in the order of: $f(s) = g(s)+ \varepsilon h(s)$, $\varepsilon>1$
- ε-suboptimal
  $cost(solution) \leq \varepsilon \cdot cost(optimal\ solution)$
- MUCH faster than A* for many problems

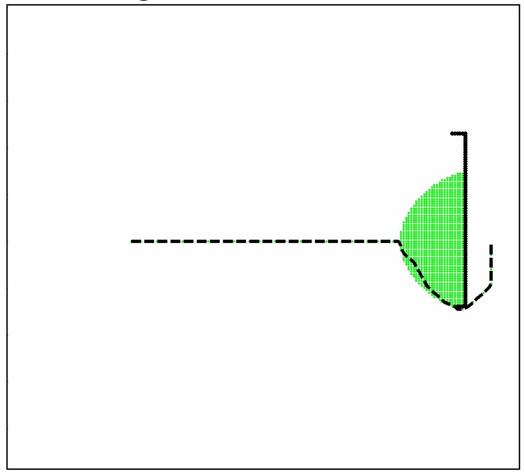
Weighted A*: Example

weighted A with ε =1 (i.e. A*)*



11,054 expansions
solution cost=168,204

weighted A with ε =10*

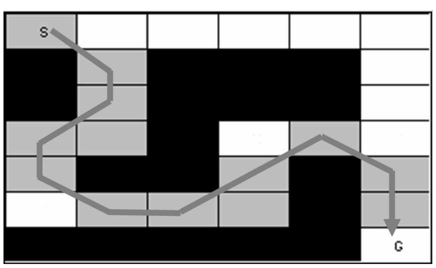


1,138 expansions
solution cost=177,876

Constructing anytime search

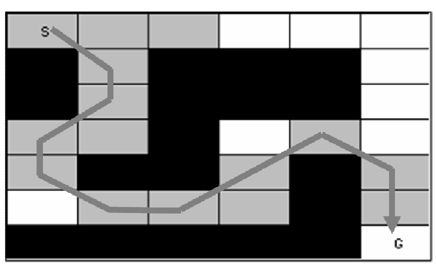
- Running a series of weighted A* searches with decreasing *ε:*

ε =2.5



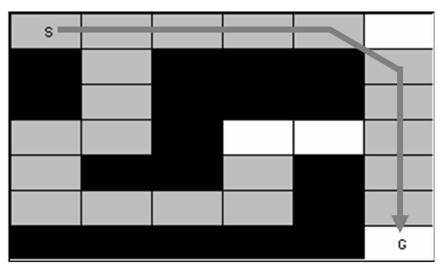
13 expansions
solution=11 moves

ε =1.5



15 expansions
solution=11 moves

ε =1.0



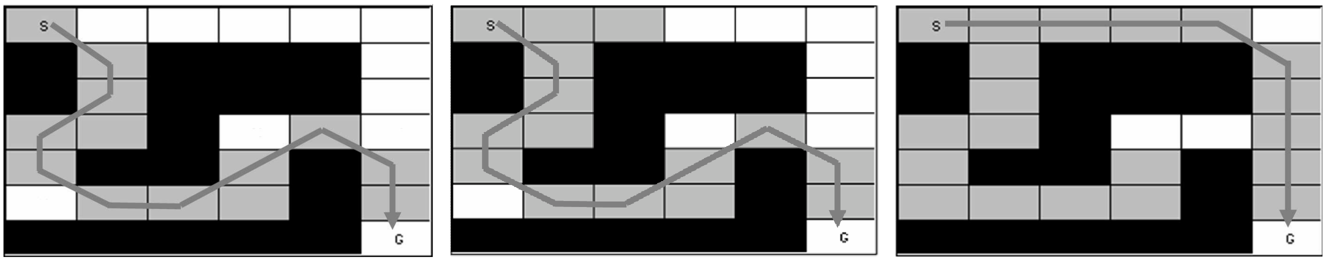
20 expansions
solution=10 moves

set ε to large value;
while $\varepsilon \geq 1$ and still has some time for planning
  run weighted A search;*
  publish current ε suboptimal solution;
  decrease ε;

Constructing anytime search

- Running a series of weighted A* searches with decreasing ε:

ε =2.5 | *ε =1.5* | *ε =1.0*



13 expansions solution=11 moves | *15 expansions solution=11 moves* | *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

ARA*: Efficient anytime search

- Runs a series of weighted A* searches with decreasing ε
- Each weighted A* search is modified to reuse previous search results
- Continues to guarantee ε suboptimality bounds

Weighted A* Search with Reuse

all *v*-values initially are infinite;

ComputePath function

while(s_{goal} is not expanded)

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* such that *s'* not in *CLOSED*

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

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

insert *s'* into *OPEN*;

Weighted A* Search with Reuse

all *v*-values initially are infinite;

v-value – the value of a state during its expansion

ComputePath function

while(s_{goal} is not expanded)

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* such that *s'* not in *CLOSED*

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

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

insert *s'* into *OPEN*;

Weighted A* Search with Reuse

all *v*-values initially are infinite;

ComputePath function

while(s_{goal} is not expanded)

  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* such that *s'* not in *CLOSED*

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

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

      insert *s'* into *OPEN*;

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

Weighted A* Search with Reuse

all *v*-values initially are infinite;

ComputePath function

while(s_{goal} is not expanded)

  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* such that *s'* not in *CLOSED*

    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 $v(s) = g(s)$

overconsistent state

consistent state

Weighted A* Search with Reuse

initialize *OPEN* with all overconsistent states;

ComputePathwithReuse function

while(s_{goal} is not expanded)

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 such that s' not in *CLOSED*

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

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

insert s' into *OPEN*;

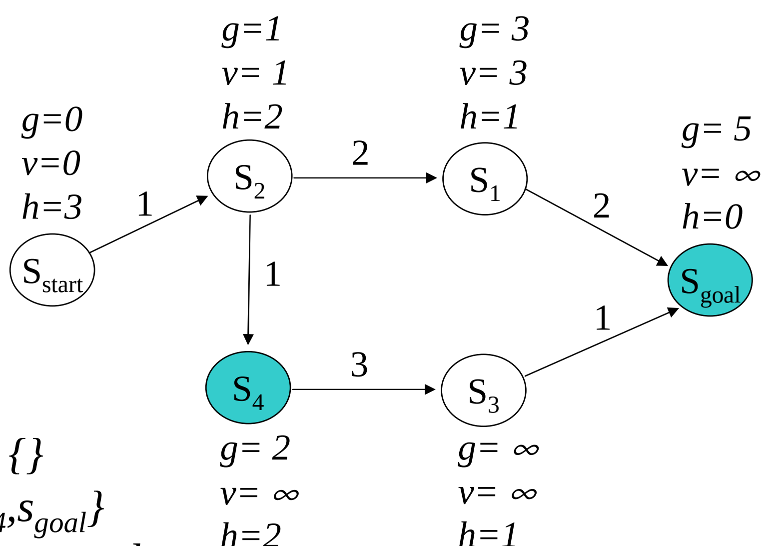
all you need to do to make it reuse old values!

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 $v(s) = g(s)$

Example: A* (ε=1) with reuse



CLOSED = {}

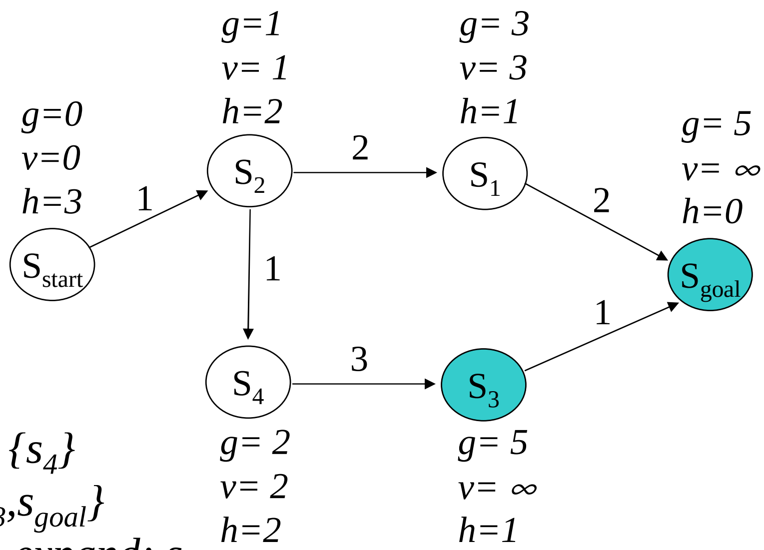
OPEN = $\{s_4, s_{goal}\}$

next state to expand: s_4

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

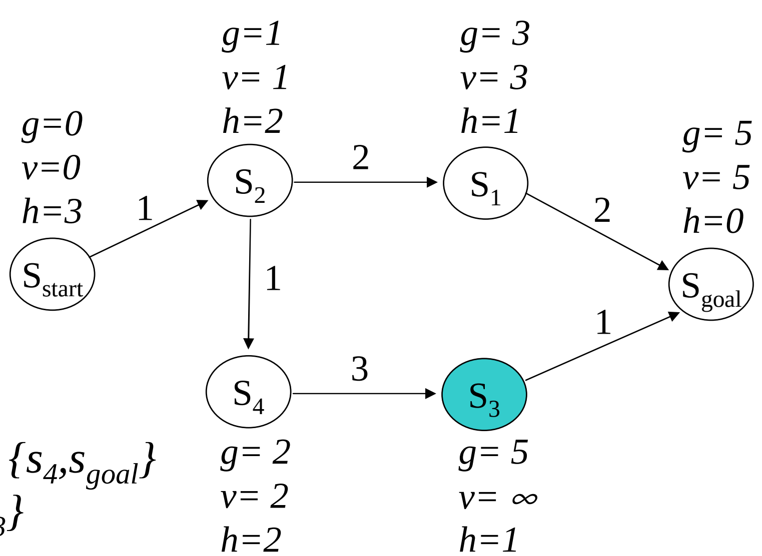
initially OPEN contains all overconsistent states

Example: A* (ε=1) with reuse



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

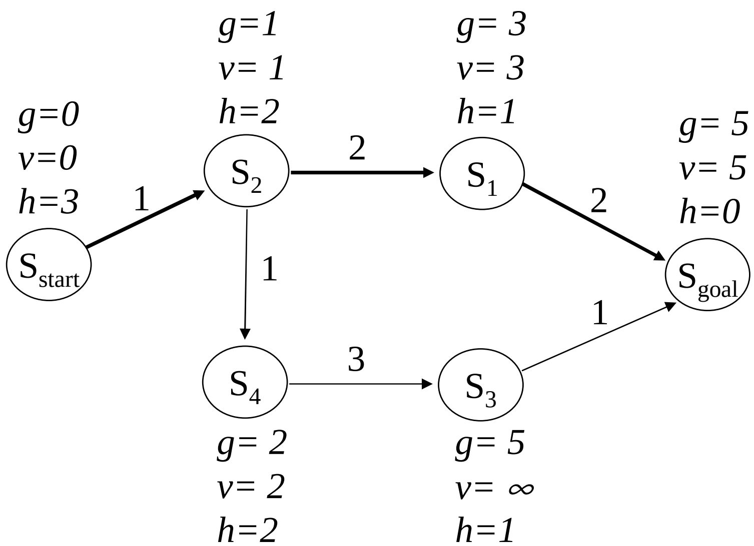
Example: A* (ε=1) with reuse



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

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

Example: A* (ε=*1)* with reuse

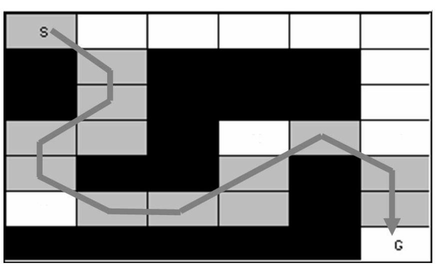


we can now compute a least-cost path

Back to Our Example

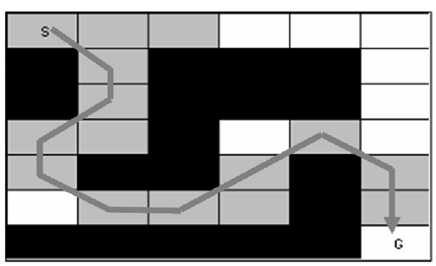
- **A series of weighted A* searches with decreasing ε:**

ε =2.5



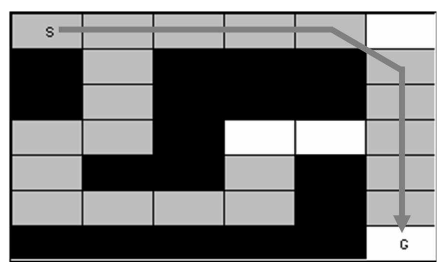
13 expansions
solution=11 moves

ε =1.5



15 expansions
solution=11 moves

ε =1.0

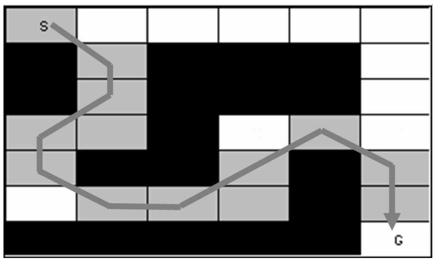


20 expansions
solution=10 moves

Back to Our Example

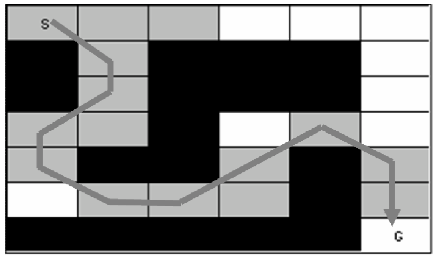
- **A series of weighted A* searches with decreasing *ε*:**

ε =2.5



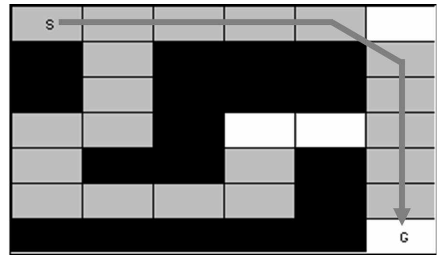
13 expansions
solution=11 moves

ε =1.5



15 expansions
solution=11 moves

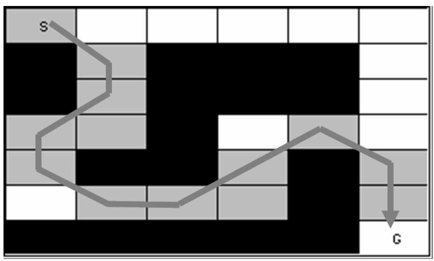
ε =1.0



20 expansions
solution=10 moves

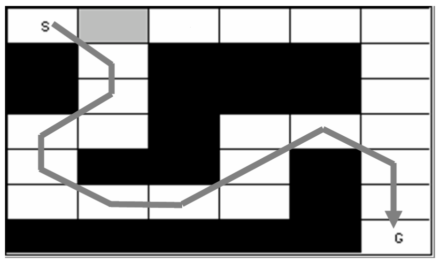
- **ARA*: a series of calls to ComputePathwithReuse with decreasing *ε*:**

ε =2.5



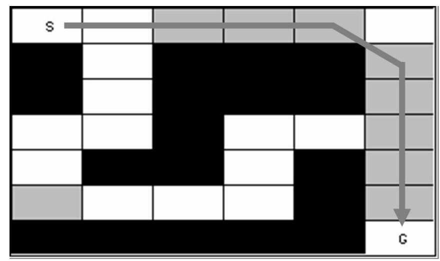
13 expansions
solution=11 moves

ε =1.5



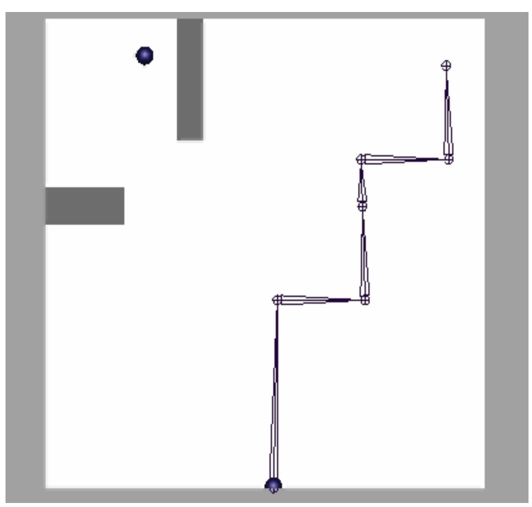
1 expansion
solution=11 moves

ε =1.0



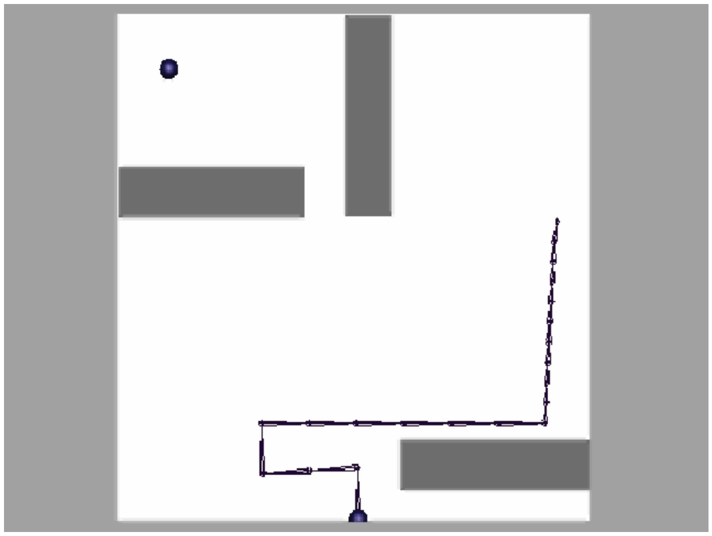
9 expansions
solution=10 moves

Planning with ARA* in High-dimensional State-spaces



after 0.05 secs of planning with ARA*

Planning with ARA* in High-dimensional State-spaces



after 90 secs of planning with ARA*

Adding Replanning Capability

- In dynamic environments edge costs change

- Can use the same ComputePathwithReuse to re-compute a path if edge costs decrease and very similar function if edge costs increase

Optimal re-planners: D* and D* Lite

set ε to 1;
until goal is reached
  ComputePathwithReuse();
  publish current ε suboptimal solution path;
  follow the path until sense something that is not in the map;
  update the corresponding edge costs;
  set s_{goal} to the current state of the agent;

Optimal re-planners: D* and D* Lite

set ε to 1;
until goal is reached
  ComputePathwithReuse();
  publish current ε suboptimal solution path;
  follow the path until sense something that is not in the map;
  update the corresponding edge costs;
  set s_{goal} to the current state of the agent;

Important detail! search is done backwards:
s_{start} = agent's goal, s_{goal} = agent's current state, all edges are reversed

Optimal re-planners: D* and D* Lite

set ε to 1;
until goal is reached
ComputePathwithReuse();
publish current ε suboptimal solution path;
follow the path until sense something that is not in the map;
update the corresponding edge costs;
set s_{goal} to the current state of the agent;

Important detail! search is done backwards:
s_{start} = agent's goal, s_{goal} = agent's current state, all edges are reversed

This way, s_{start} always remains the same and g-values are more likely to remain the same in between two calls to ComputePathwithReuse

Optimal re-planners: D* and D* Lite

set ε to 1;
until goal is reached
ComputePathwithReuse();
publish current ε suboptimal solution path;
follow the path until sense something that is not in the map;
update the corresponding edge costs;
set s_{goal} to the current state of the agent;

Important detail! search is done backwards:
s_{start} = agent's goal, s_{goal} = agent's current state, all edges are reversed

This way, s_{start} always remains the same and g-values are more likely to remain the same in between two calls to ComputePathwithReuse

why?

why care?

D* & D* Lite: Example

initial knowledge and initial goal distances

14	13	12	11	10	9	8	7	6	6	6	6	6	6	6	6	6	6
14	13	12	11	10	9	8	7	6	5	5	5	5	5	5	5	5	5
14	13	12	11	10	9	8	7	6	5	4	4	4	4	4	4	4	4
14	13	12	11	10	9	8	7	6	5	4	3	3	3	3	3	3	3
14	13	12	11	10	9	8	7	6	5	4	3	2	2	2	2	2	3
14	13	12	11	10	9	8	7	6	5	4	3	2	1	1	1	2	3
14	13	12	11		9		7	6	5	4	3	2	1	s_{goal}	1	2	3
					9				5	4	3	2	1	1	1	2	3
14	13	12	11	10	9	8	7	6	5	4	3	2	2	2	2	2	3
14	13	12	11	10	9				5	4	3	3	3	3	3	3	3
14	13	12	11	10	10		7	6	5	4	4	4	4	4	4	4	4
14	13	12	11	11	11		7	6	5	5	5	5	5	5	5	5	5
14	13	12	12	12	12		7	6	6	6	6	6	6	6	6	6	6
					13		7	7	7	7	7	7	7	7	7	7	7
18	s_{start}	16	15	14	14		8	8	8	8	8	8	8	8	8	8	8

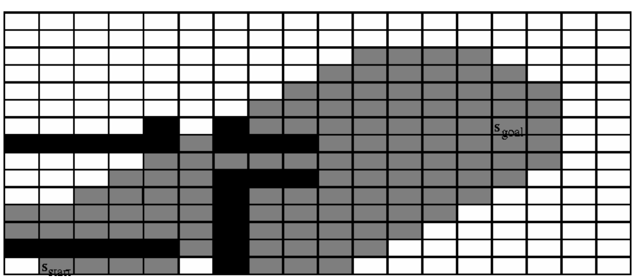
knowledge and goal distances after the robot moves

cells in gray have g-values changed

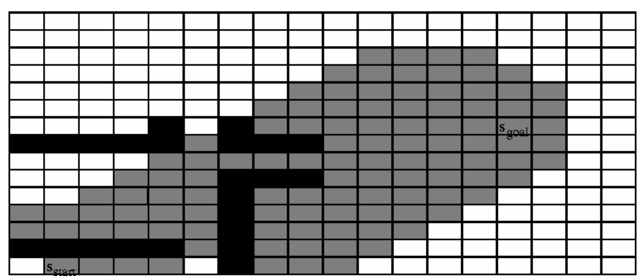
14	13	12	11	10	9	8	7	6	6	6	6	6	6	6	6	6	6
14	13	12	11	10	9	8	7	6	5	5	5	5	5	5	5	5	5
14	13	12	11	10	9	8	7	6	5	4	4	4	4	4	4	4	4
14	13	12	11	10	9	8	7	6	5	4	3	3	3	3	3	3	3
14	13	12	11	10	9	8	7	6	5	4	3	2	2	2	2	2	3
14	13	12	11	10	9	8	7	6	5	4	3	2	1	1	1	2	3
14	13	12	11		9		7	6	5	4	3	2	1	s_{goal}	1	2	3
					10				5	4	3	2	1	1	1	2	3
15	14	13	12	11	11		7	6	5	4	3	2	2	2	2	2	3
15	14	13	12	12	s_{start}				5	4	3	3	3	3	3	3	3
15	14	13	13	13	13		7	6	5	4	4	4	4	4	4	4	4
15	14	14	14	14	14		7	6	5	5	5	5	5	5	5	5	5
15	15	15	15	15	15		7	6	6	6	6	6	6	6	6	6	6
					16		7	7	7	7	7	7	7	7	7	7	7
21	20	19	18	17	17		8	8	8	8	8	8	8	8	8	8	8

D* & D* Lite: Example

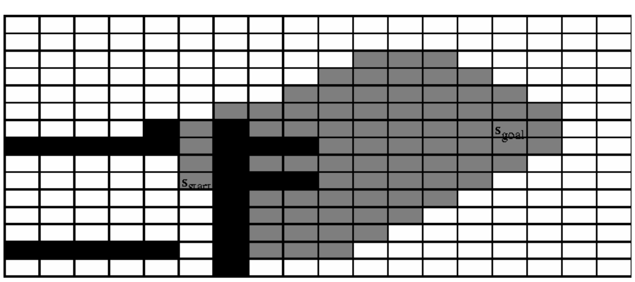
initial A* search



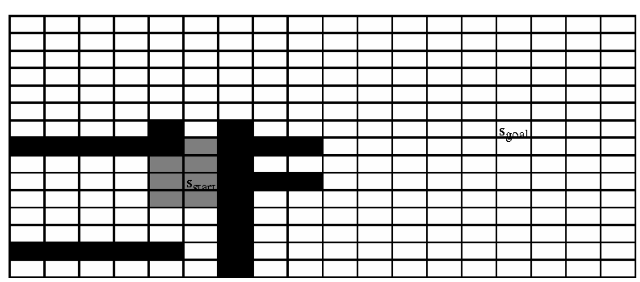
initial D* Lite search



second A* search



second D* Lite search



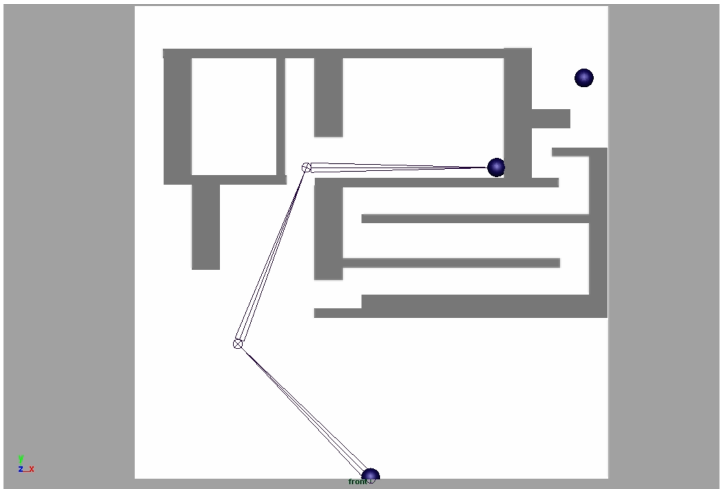
Anytime re-planner: Anytime D*

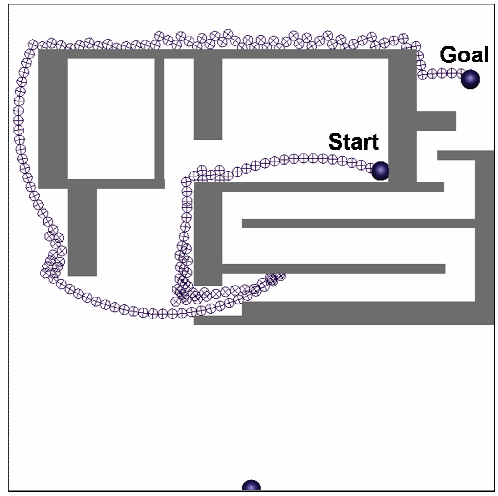
set ε to ***large value;***
until goal is reached
  ComputePathwithReuse();
  publish current ε suboptimal solution path;
  follow the path until sense something that is not in the map;
  update the corresponding edge costs;
  set s_{goal} to the current state of the agent;
  if significant changes were observed
    increase ε or replan from scratch;
  else
    decrease ε;

Anytime re-planner: Anytime D*

set ε to ***large value;***
until goal is reached
  ComputePathwithReuse();
  publish current ε suboptimal solution path;
  follow the path until sense something that is not in the map;
  update the corresponding edge costs;
  set s_{goal} to the current state of the agent;
  if significant changes were observed
    increase ε or replan from scratch; *why?*
  else
    decrease ε;

Planning with Anytime D*





- 3 DOF robotic arm manipulating an end-effector through dynamic environment
- 1 sec of deliberation (improving and/or replanning) in between each step
- Initially, $\varepsilon = 20$

Summary

- Planning is often a repeated process and needs to be fast
  - dynamic environments
  - inaccurate initial model
  - errors in the position of the agent

- Family of A*-based planners:
  - ARA*
    - anytime A* search
    - outputs ε suboptimal solutions
    - can be used under time constraints

  - D* and D* Lite
    - incremental A* search
    - computes optimal solutions by reusing previous search efforts
    - can often drastically speed up repeated planning

  - Anytime D* (AD*)
    - anytime incremental A* search
    - outputs ε suboptimal solutions
    - can be used under time constraints
    - can often drastically speed up repeated planning

  - all based on the ComputePathwithReuse function