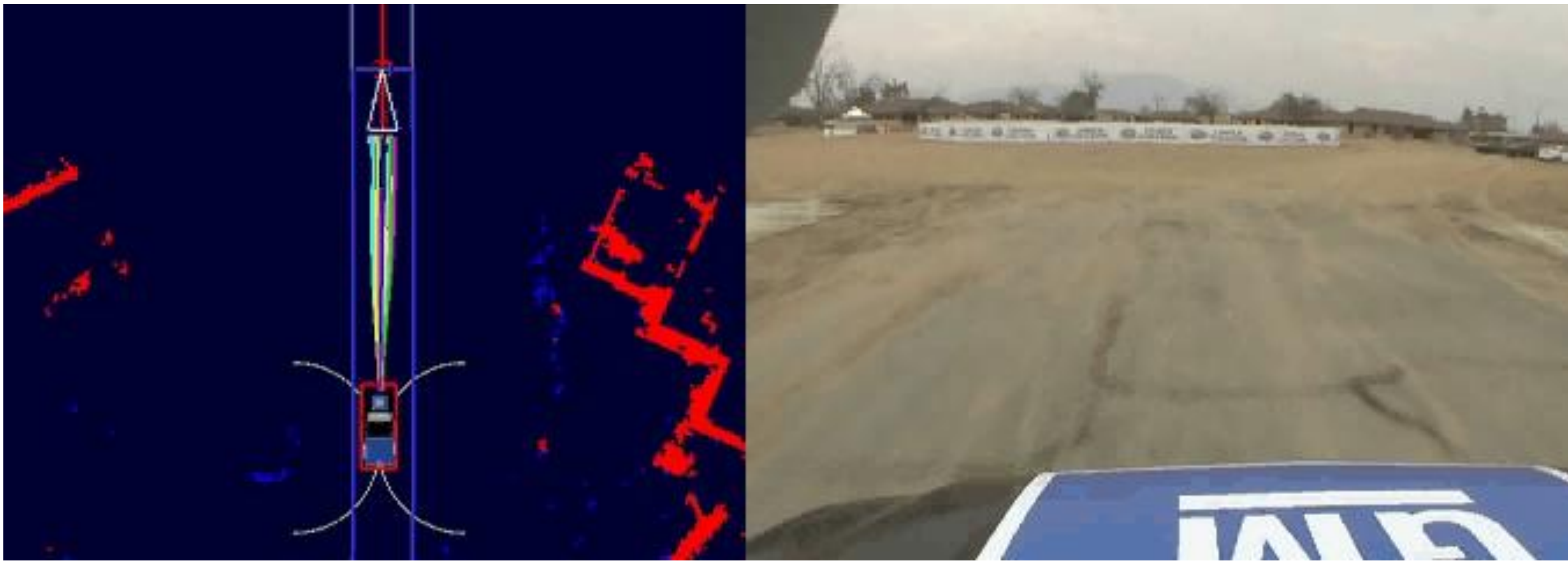
Real-time Planning and Re-planning I: Incremental & Anytime Search

Maxim Likhachev

Carnegie Mellon University

Example of Real-time (Re-)planning

- Using anytime incremental A* (Anytime D*) in Urban Challenge



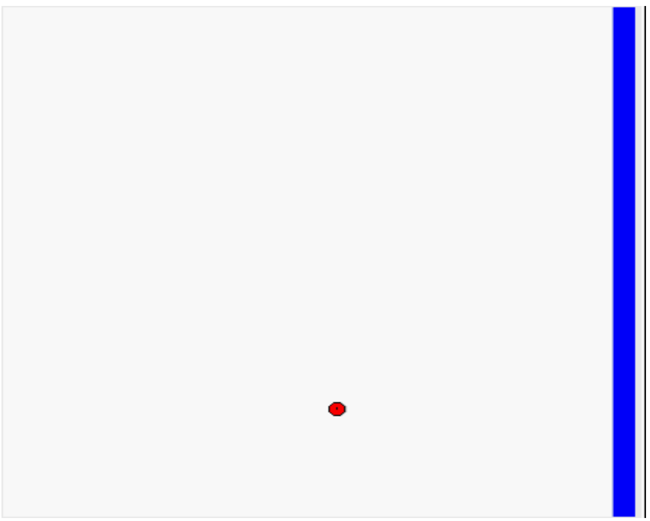
Tartanracing, CMU

Dynamic and Partially-known Environments

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

ATRV navigating initially-unknown environment

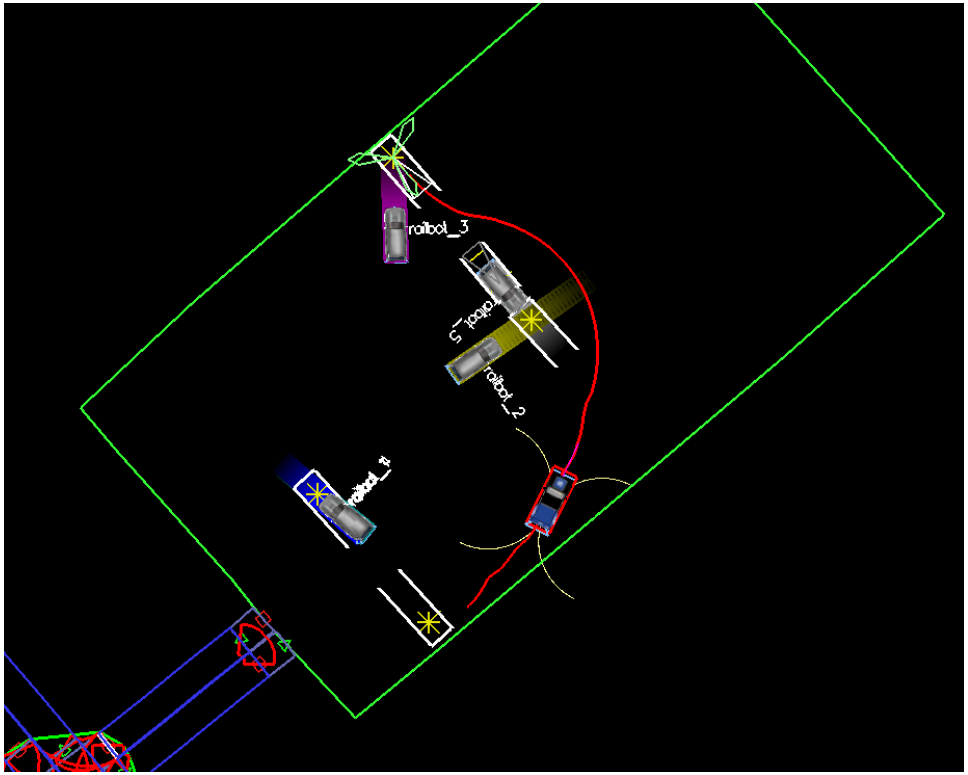
planning map and path



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



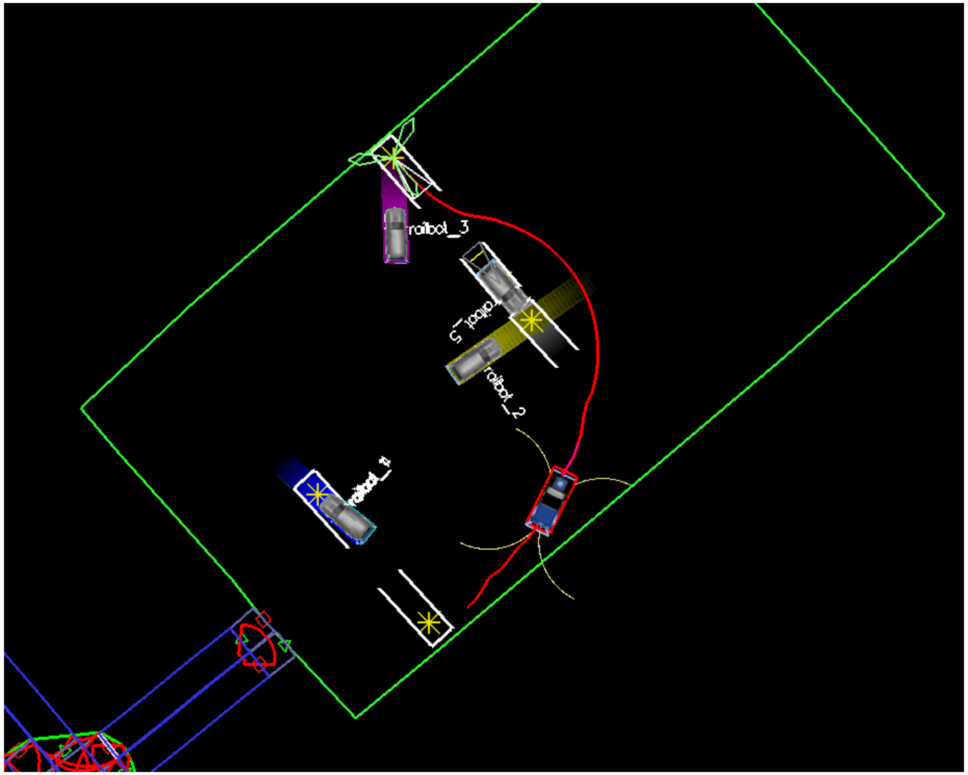
Tartanracing, CMU

Dynamic and Partially-known Environments

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

Other reasons for re-planning?

planning in dynamic environments

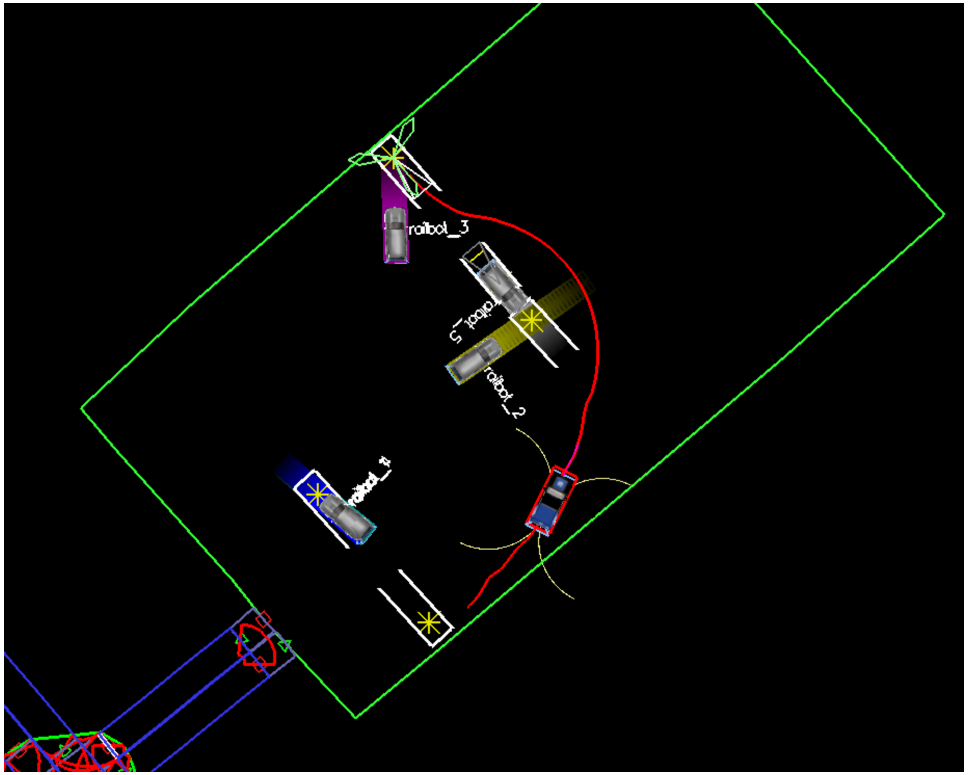


Tartanracing, CMU

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

- Need to re-plan fast!
- Two ways to help with this requirement
  - anytime planning
  - incremental planning

this class:
incremental version of A and how it can be used for anytime and incremental planning*

Motivation for Incremental Version of A*

- Reuse state values from previous searches

cost of least-cost paths to s_{goal} initially

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

cost of least-cost paths to s_{goal} after the door turns out to be closed

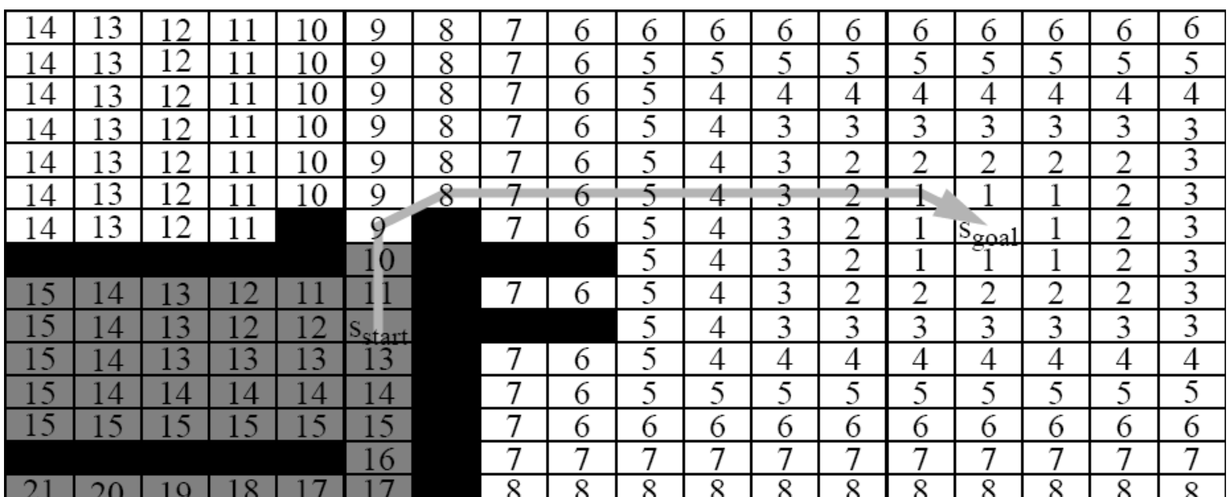
14	13	12	11	10	9	8	7	6	6	6	6	6	6	6	6	6	6
14	13	12	11	10	9	8	7	6	5	5	5	5	5	5	5	5	5
14	13	12	11	10	9	8	7	6	5	4	4	4	4	4	4	4	4
14	13	12	11	10	9	8	7	6	5	4	3	3	3	3	3	3	3
14	13	12	11	10	9	8	7	6	5	4	3	2	2	2	2	2	3
14	13	12	11	10	9	8	7	6	5	4	3	2	1	1	1	2	3
14	13	12	11		9		7	6	5	4	3	2	1	s_{goal}	1	2	3
					10		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

Motivation for Incremental Version of A*

- Reuse state values from previous searches

cost of least-cost paths to s_{goal} initially

cost of least-cost paths to s_{goal} after the door turns out to be closed



Motivation for Incremental Version of A*

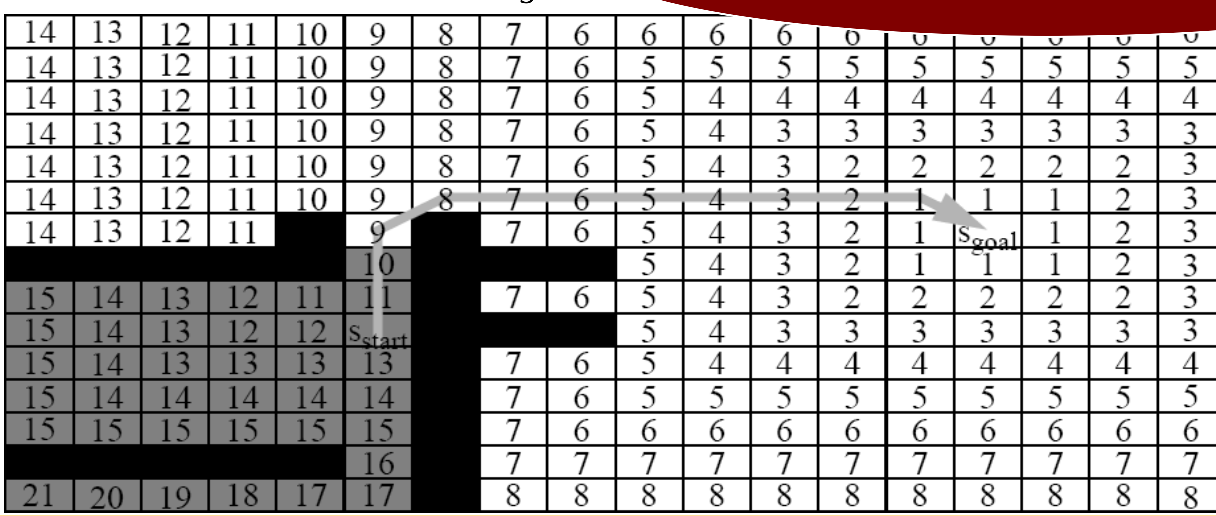
- Reuse state values from previous searches

cost of least-cost paths to s_{goal} initially

These costs are optimal g-values if search is done backwards

*Can we reuse these g-values from one search to another? – incremental A**

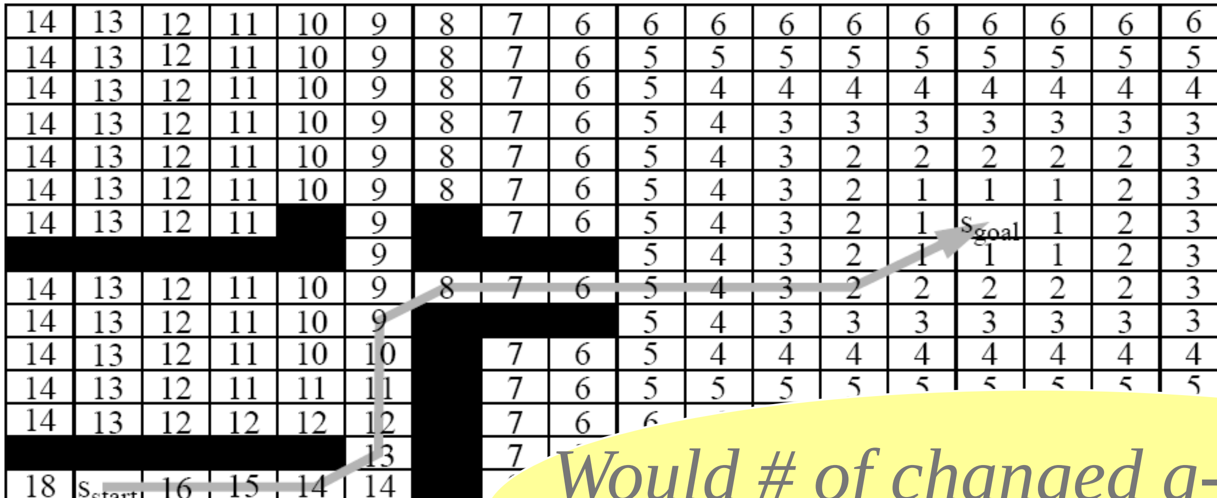
cost of least-cost paths to s_{goal}



Motivation for Incremental Version of A*

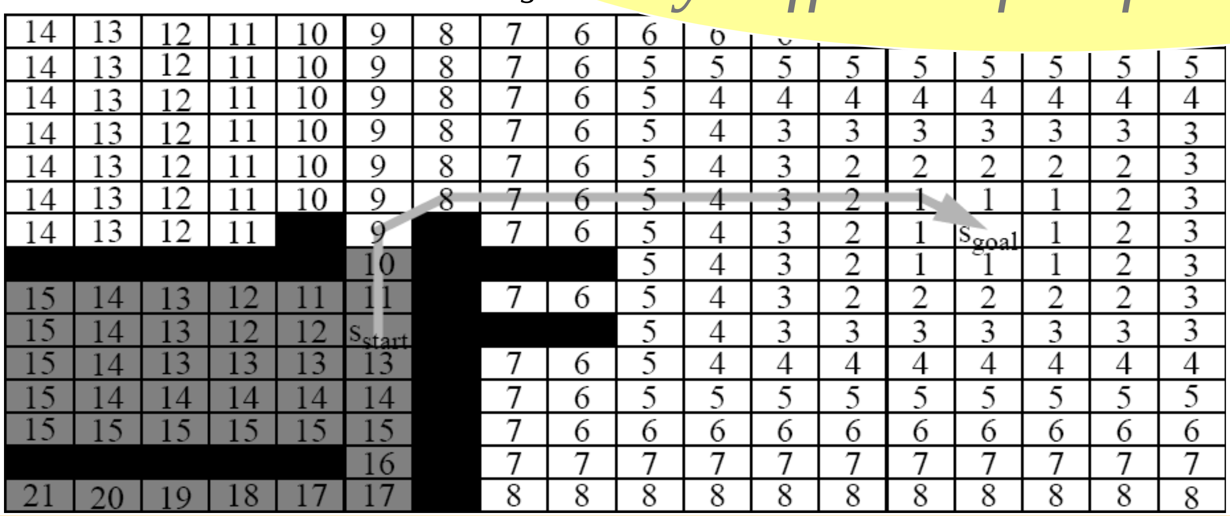
- Reuse state values from previous searches

cost of least-cost paths to s_{goal} initially



Would # of changed g-values be very different for forward A?*

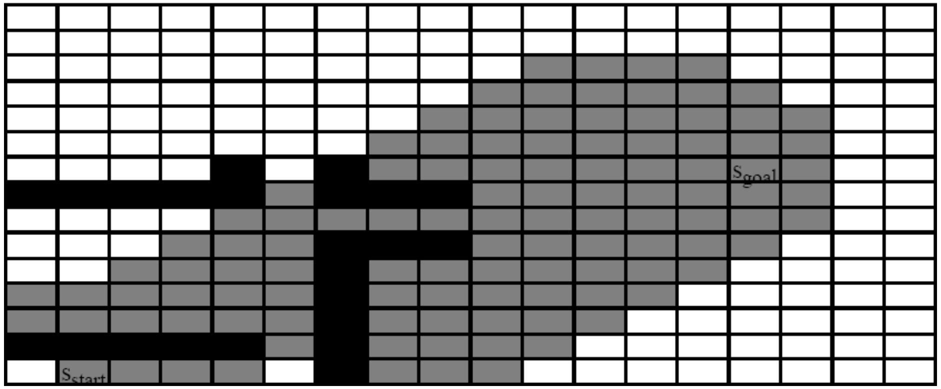
cost of least-cost paths to s_{go}



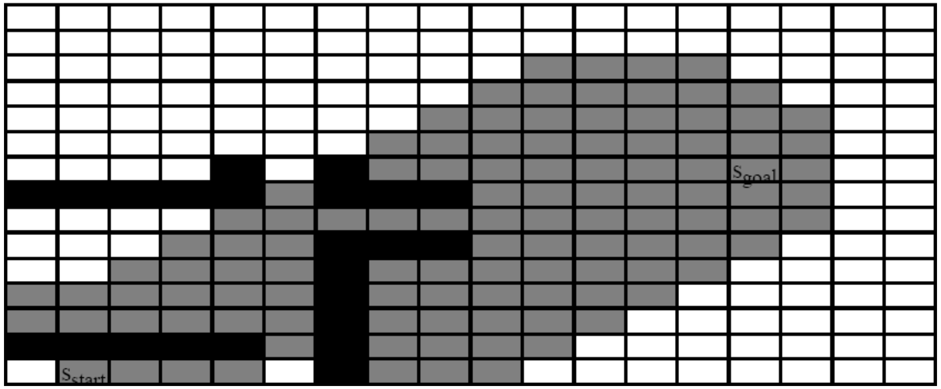
Incremental Version of A*

- Reuse state values from previous searches

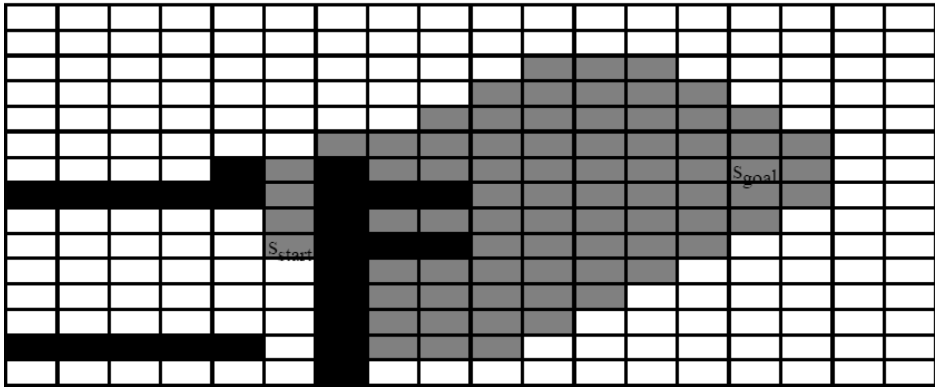
*initial search by backwards A**



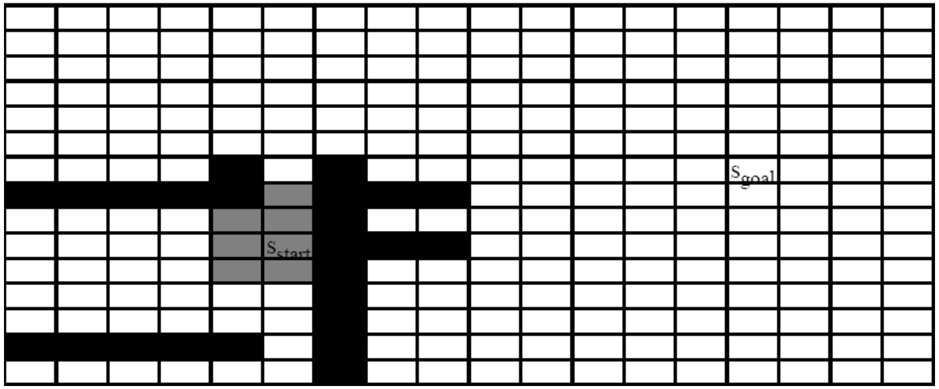
initial search by D Lite*



*second search by backwards A**



second search by D Lite*



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*;

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*;

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

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

- 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 $v(s) = g(s)$

overconsistent state

consistent state

Why?

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*;

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

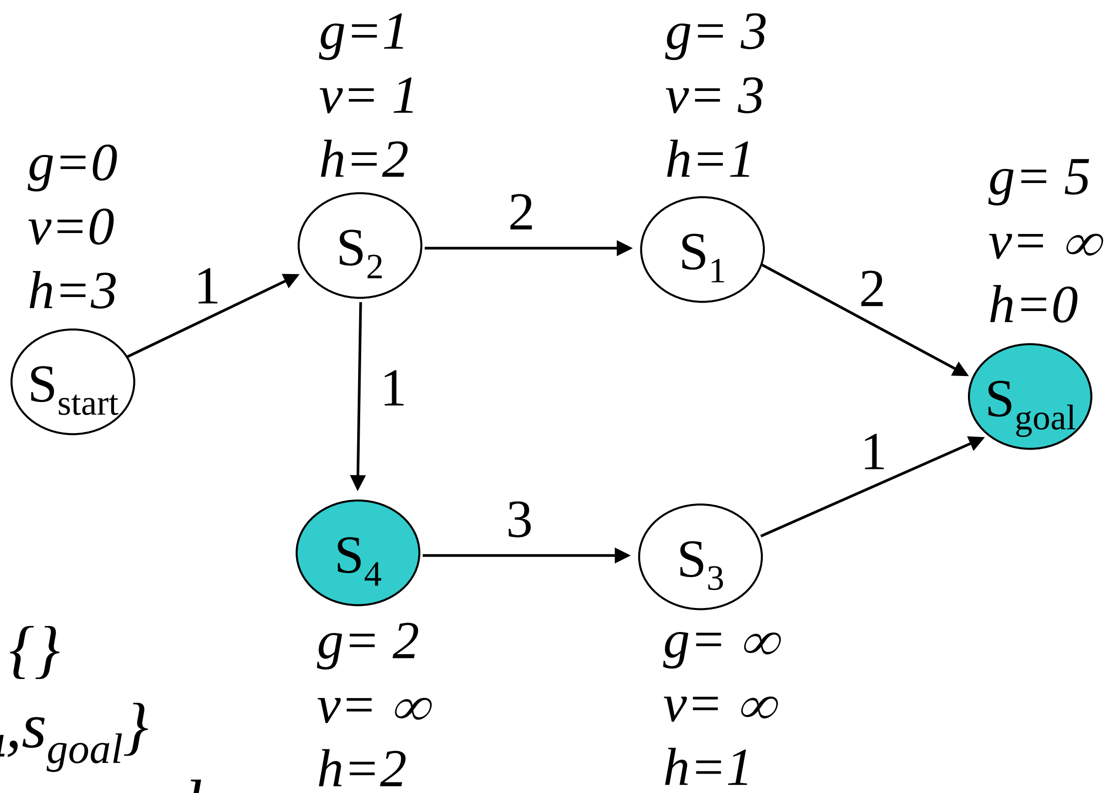
      $g(s') = g(s) + c(s,s');$

      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



CLOSED = {}

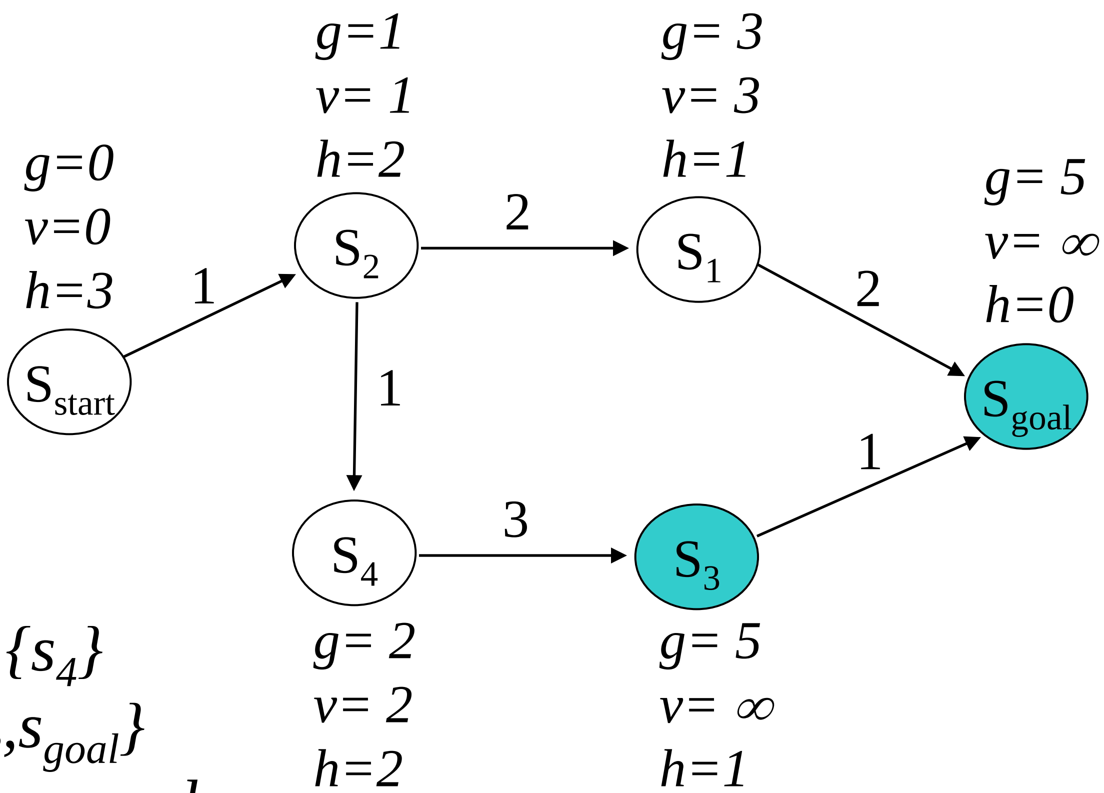
OPEN = $\{s_4, s_{goal}\}$

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

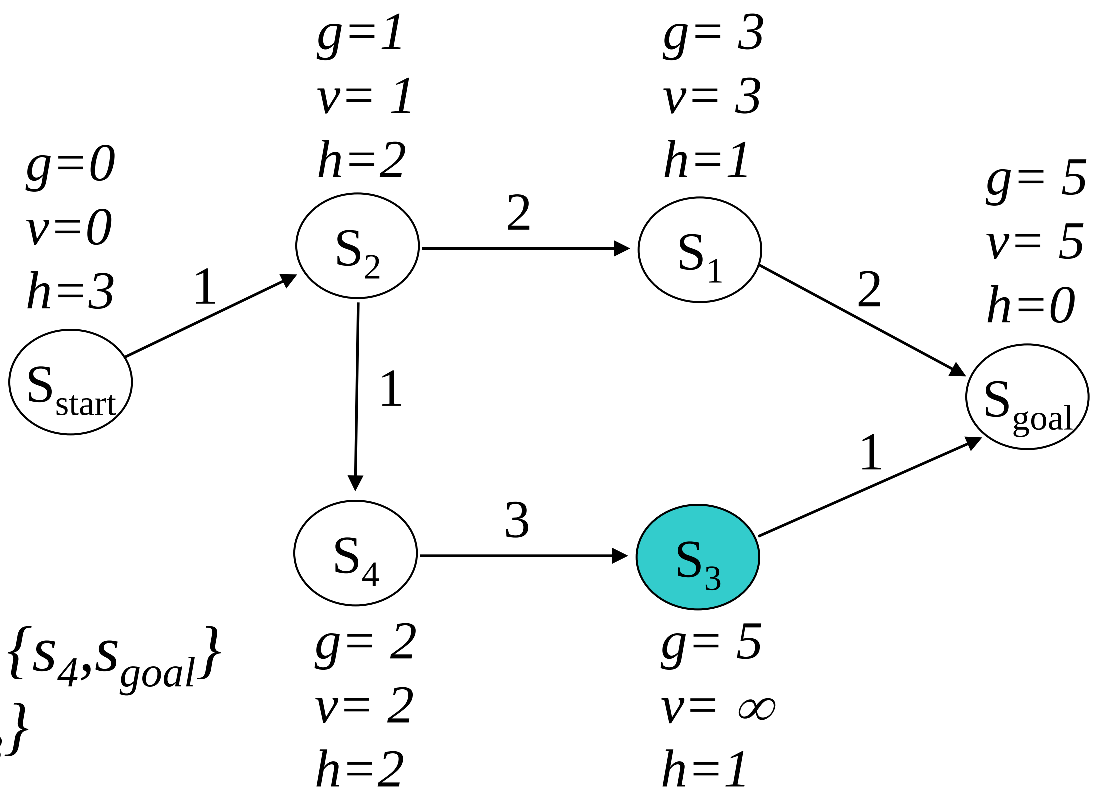


CLOSED = $\{s_4\}$

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

next state to expand: s_{goal}

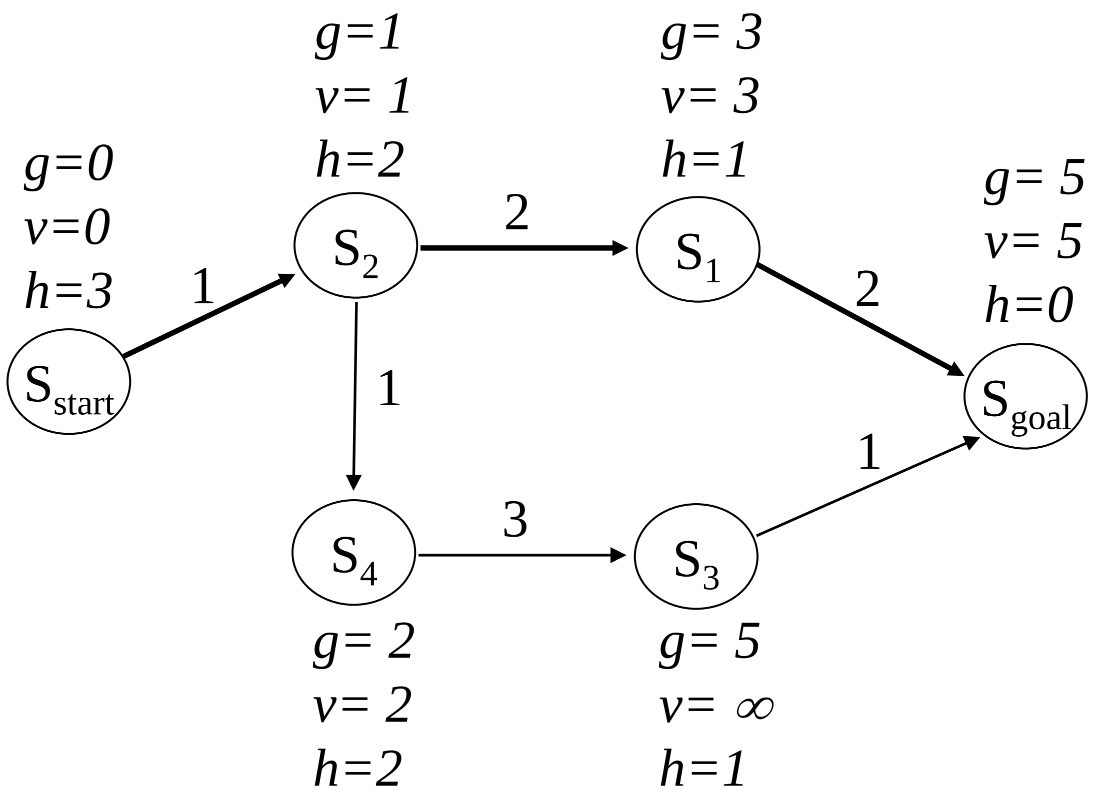
A* with Reuse of State Values



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

Why didn't we do it for A?*

Anytime Repairing A* (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 $\varepsilon \geq 1$
  $CLOSED = \{\}$;
  ComputePathwithReuse();
  publish current ε suboptimal solution;
  decrease ε;
  initialize *OPEN* with 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 $\varepsilon \geq 1$
  $CLOSED = \{\}$;
  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 *Why?*

ARA*

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

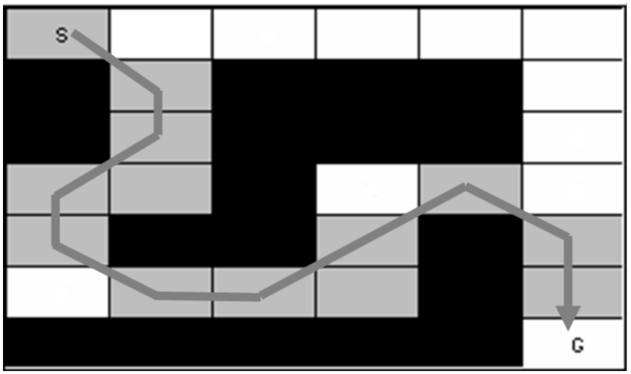
set ε to large value;
$g(s_{start}) = 0$; v-values of all states are set to infinity; $OPEN = \{s_{start}\}$;
while $\varepsilon \geq 1$
    $CLOSED = \{\}$; $INCONS = \{\}$;
    ComputePathwithReuse();
    publish current ε suboptimal solution;
    decrease ε;
    initialize $OPEN = OPEN \cup INCONS$;

all overconsistent states (exactly what we need!)

ARA*

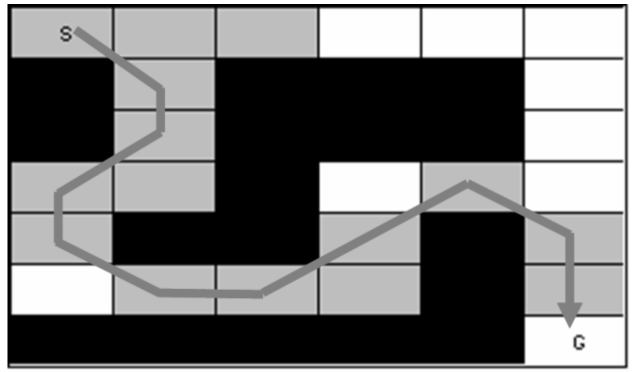
- **A series of weighted A* searches**

ε =2.5



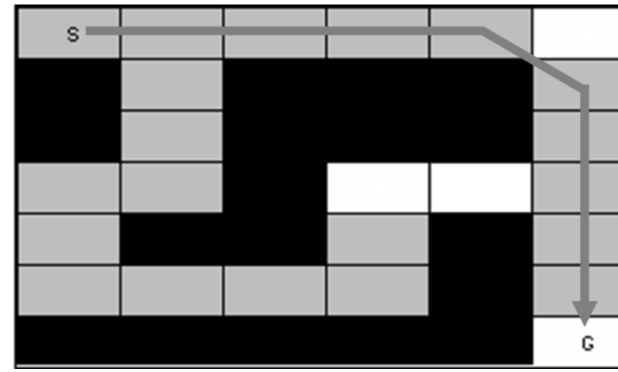
13 expansions
solution=11 moves

ε =1.5



15 expansions
solution=11 moves

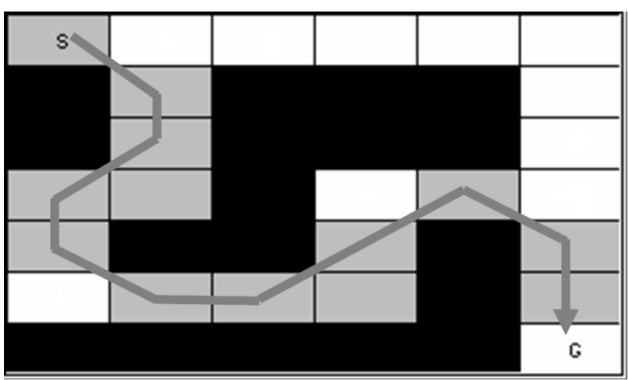
ε =1.0



20 expansions
solution=10 moves

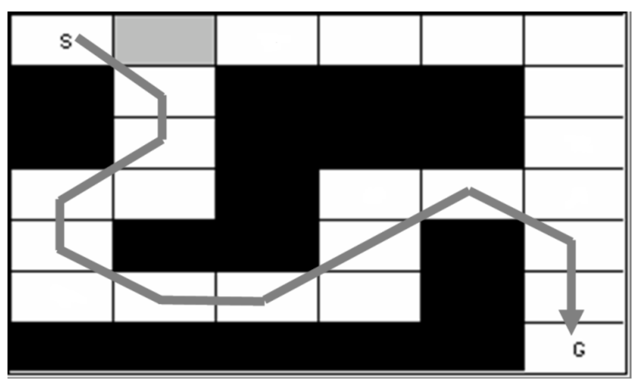
- **ARA***

ε =2.5



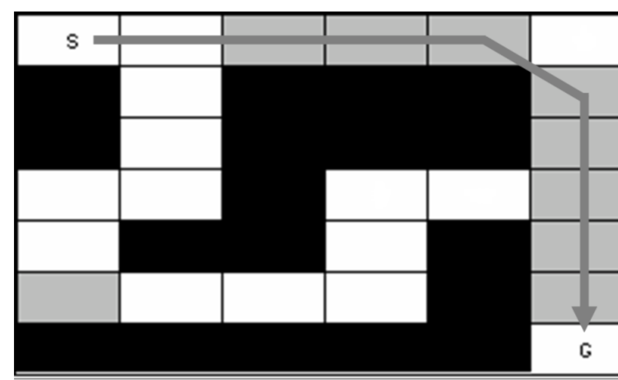
13 expansions
solution=11 moves

ε =1.5



1 expansion
solution=11 moves

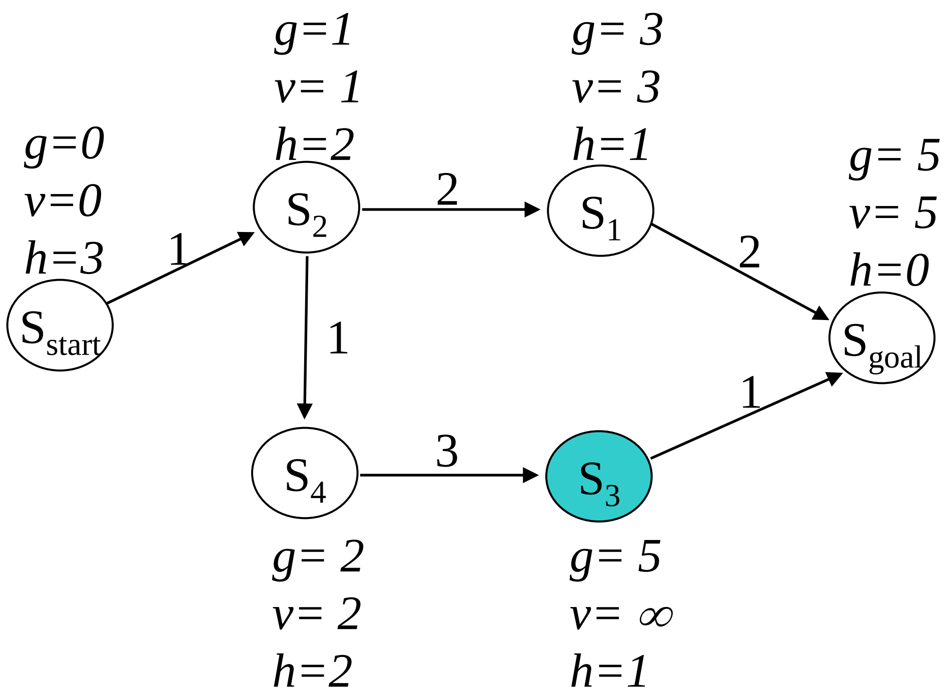
ε =1.0



9 expansions
solution=10 moves

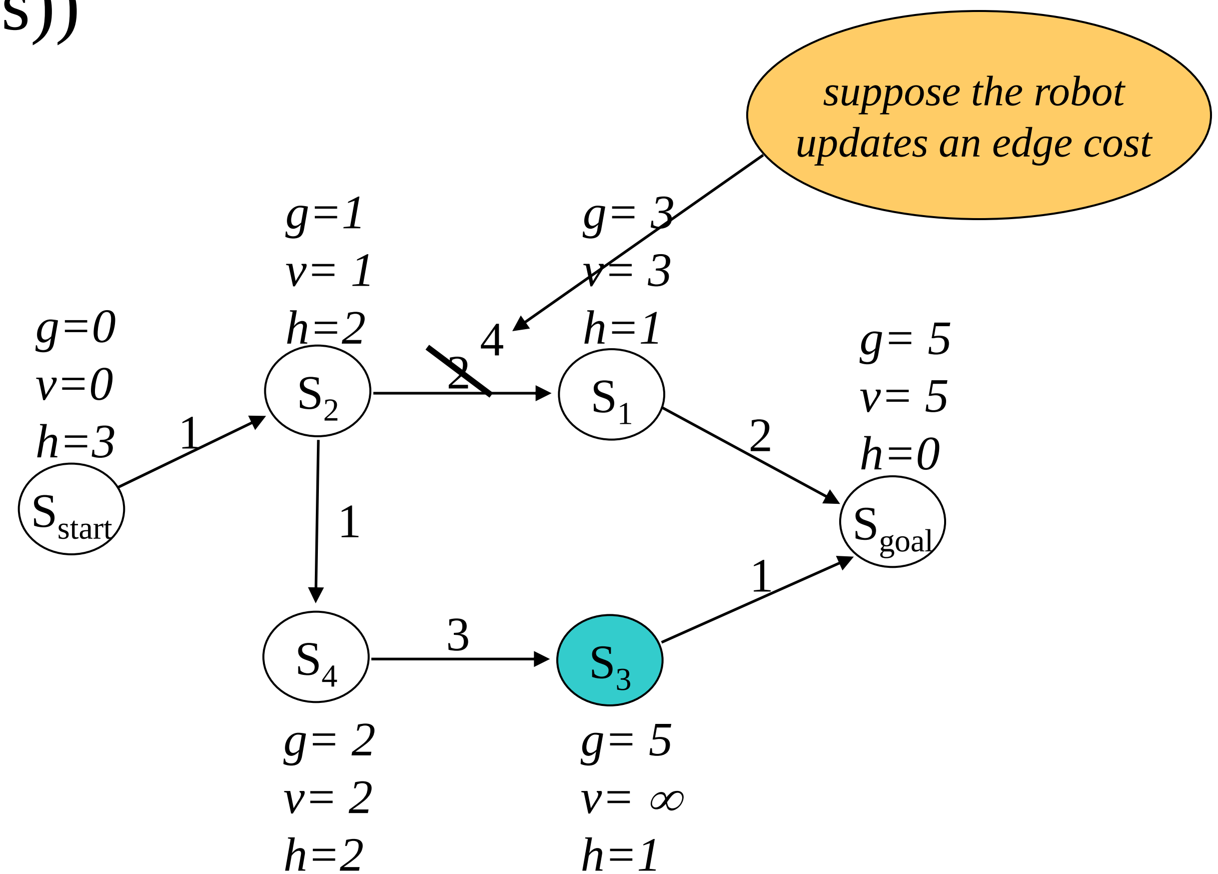
A* with Reuse of State Values

- So far, ComputePathwithReuse() could only deal with states whose $v(s) \geq g(s)$ (overconsistent or consistent)
- Edge cost increases may introduce underconsistent states ($v(s) < g(s)$)



A* with Reuse of State Values

- So far, ComputePathwithReuse() could only deal with states whose $v(s) \geq g(s)$ (overconsistent or consistent)
- Edge cost increases may introduce underconsistent states ($v(s) < g(s)$)

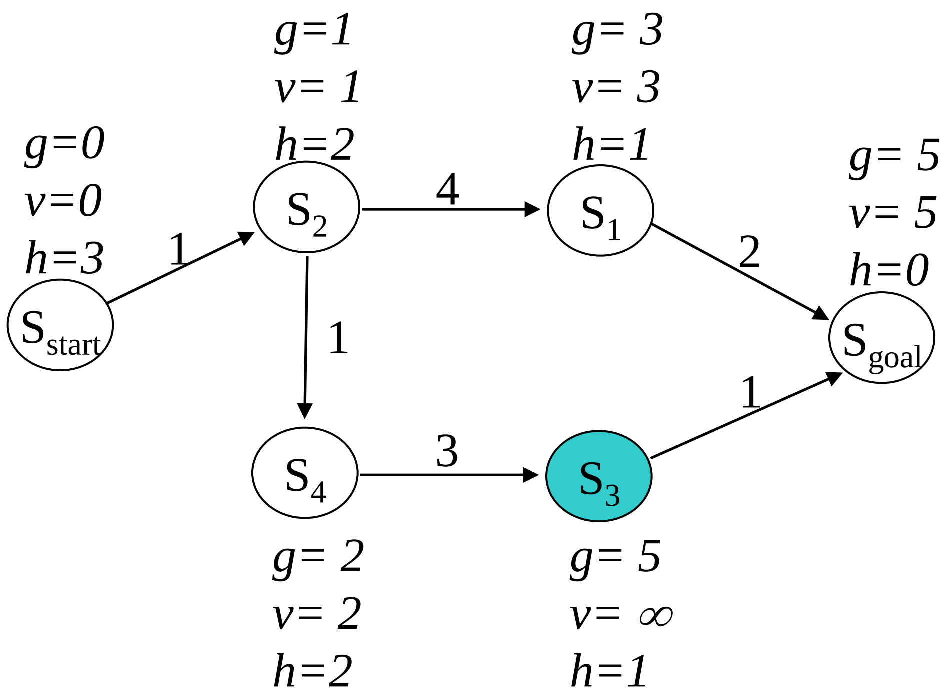


A* with Reuse of State Values

- Edge cost increases may introduce underconsistent states ($v(s) < g(s)$)

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

need to update $g(s_1)$



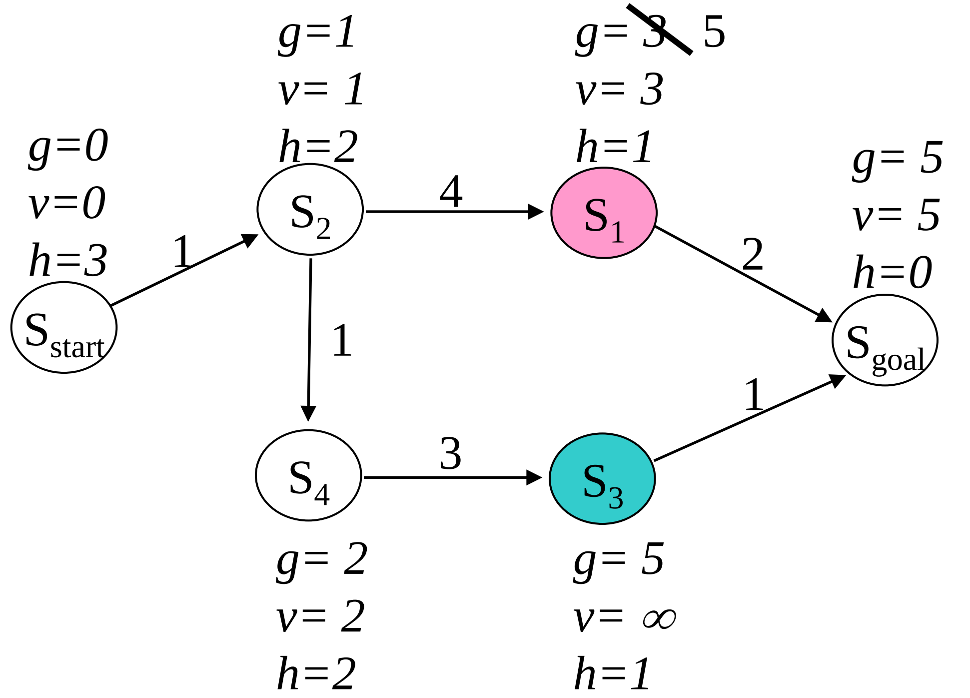
A* with Reuse of State Values

- Edge cost increases may introduce underconsistent states (v(s) < g(s))

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

need to update $g(s_1)$

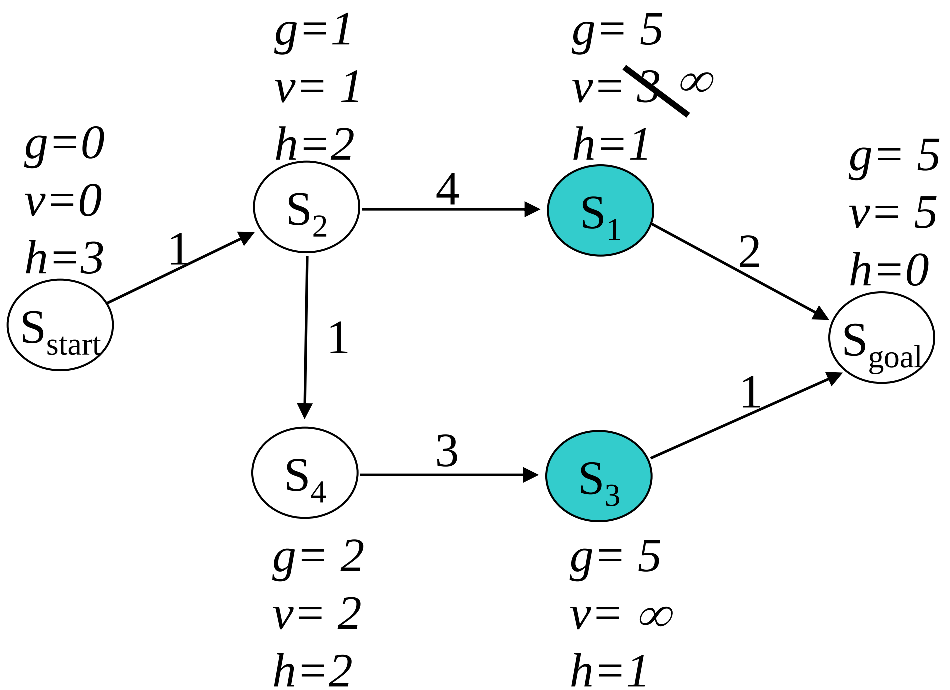
$v(s_1) < g(s_1)$



A* with Reuse of State Values

- Edge cost increases may introduce underconsistent states (v(s) < g(s))
- Fix these by setting $v(s) = \infty$

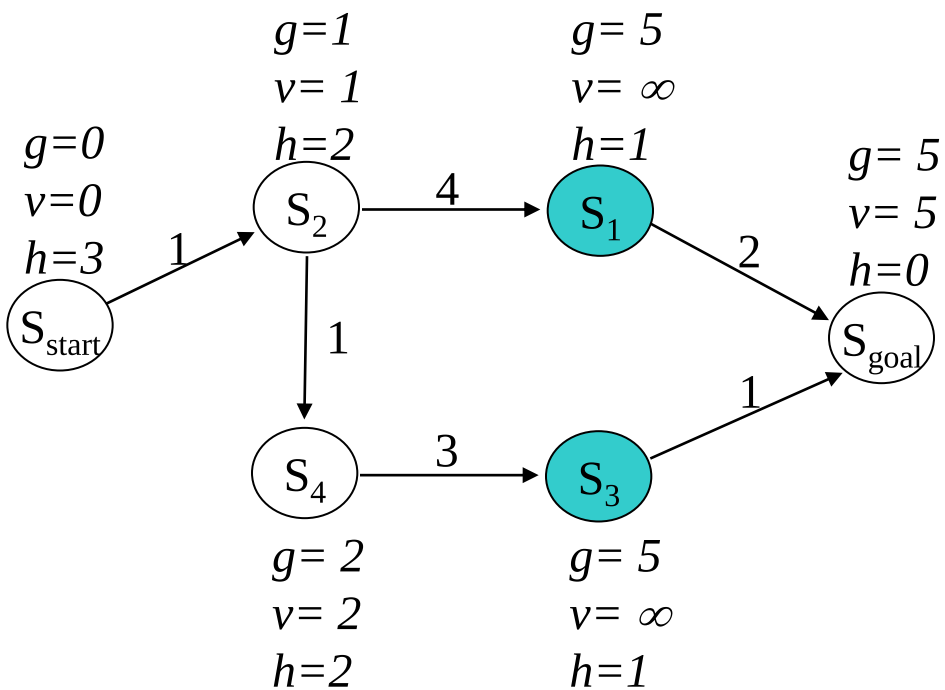
ComputePathwithReuse invariant:
$g(s') = \min_{s'' \in pred(s')} v(s'') + c(s'',s')$



A* with Reuse of State Values

- Edge cost increases may introduce underconsistent states (v(s) < g(s))
- Fix these by setting $v(s) = \infty$
- Makes *s* overconsistent or consistent $v(s) \geq g(s)$

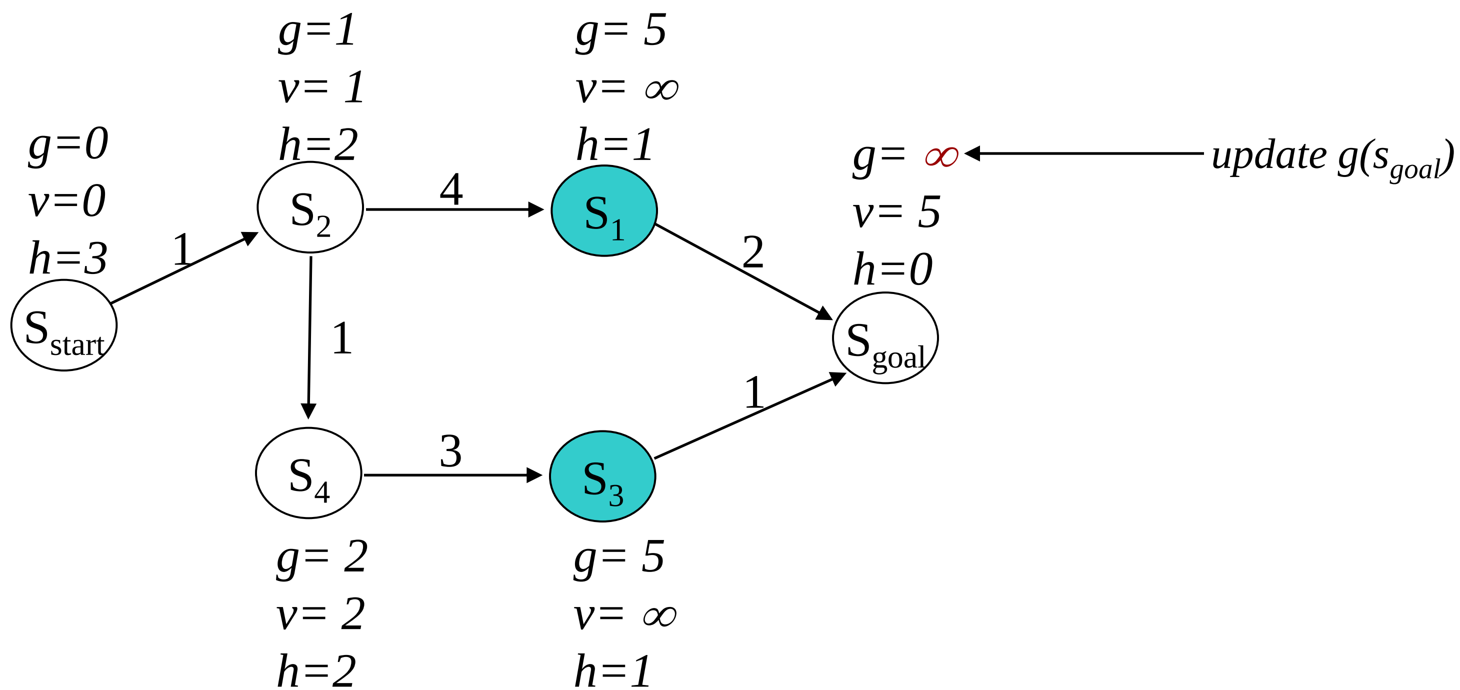
ComputePathwithReuse invariant:
$g(s') = \min_{s'' \in pred(s')} v(s'') + c(s'',s')$



A* with Reuse of State Values

- Edge cost increases may introduce underconsistent states (v(s) < g(s))
- Fix these by setting *v(s)* = ∞
- Makes *s* overconsistent or consistent *v(s)* $\geq$ *g(s)*
- Propagate the changes

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

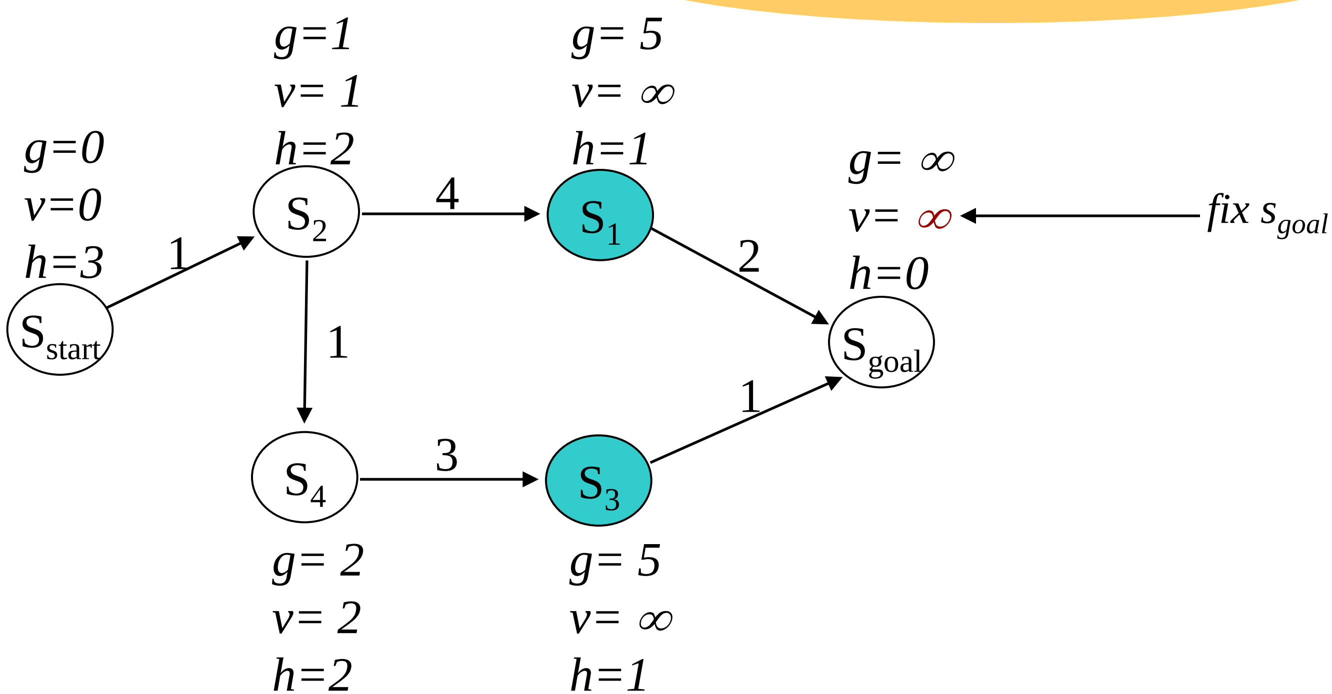


A* with Reuse of State Values

- Edge cost increases may introduce underconsistent states (v(s) < g(s))
- Fix these by setting $v(s) = \infty$
- Makes *s* overconsistent or consistent $v(s) \geq g(s)$
- Propagate the changes

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

no more underconsistent states!

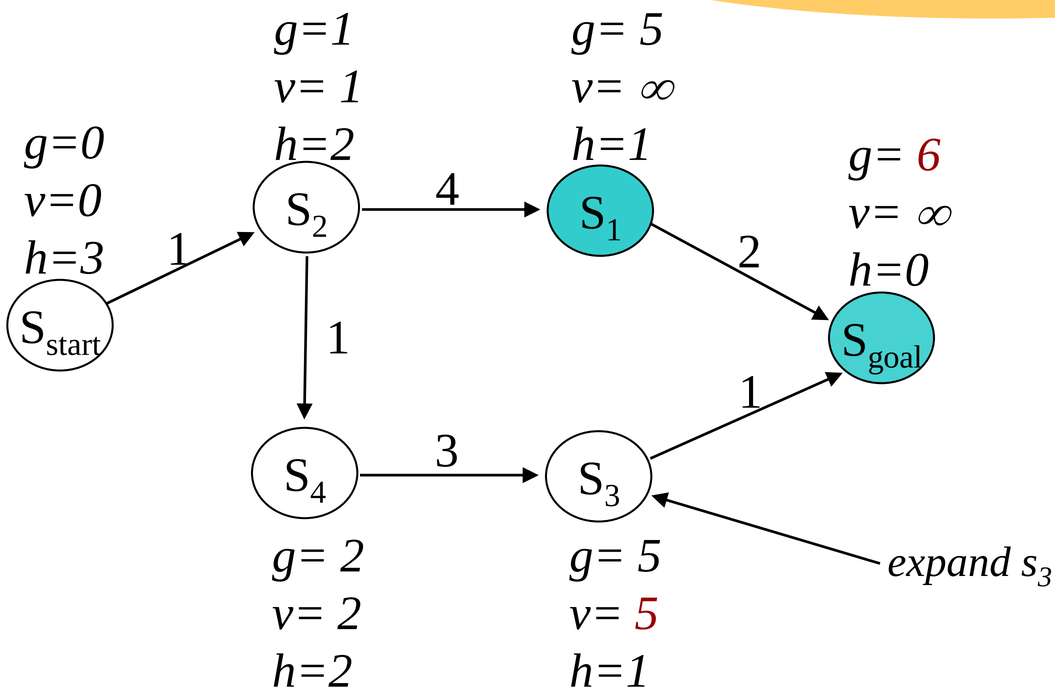


A* with Reuse of State Values

- Edge cost increases may introduce underconsistent states (v(s) < g(s))
- Fix these by setting *v(s)* = ∞
- Makes *s* overconsistent or consistent *v(s)* ≥ *g(s)*
- Propagate the changes

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

no more underconsistent states!

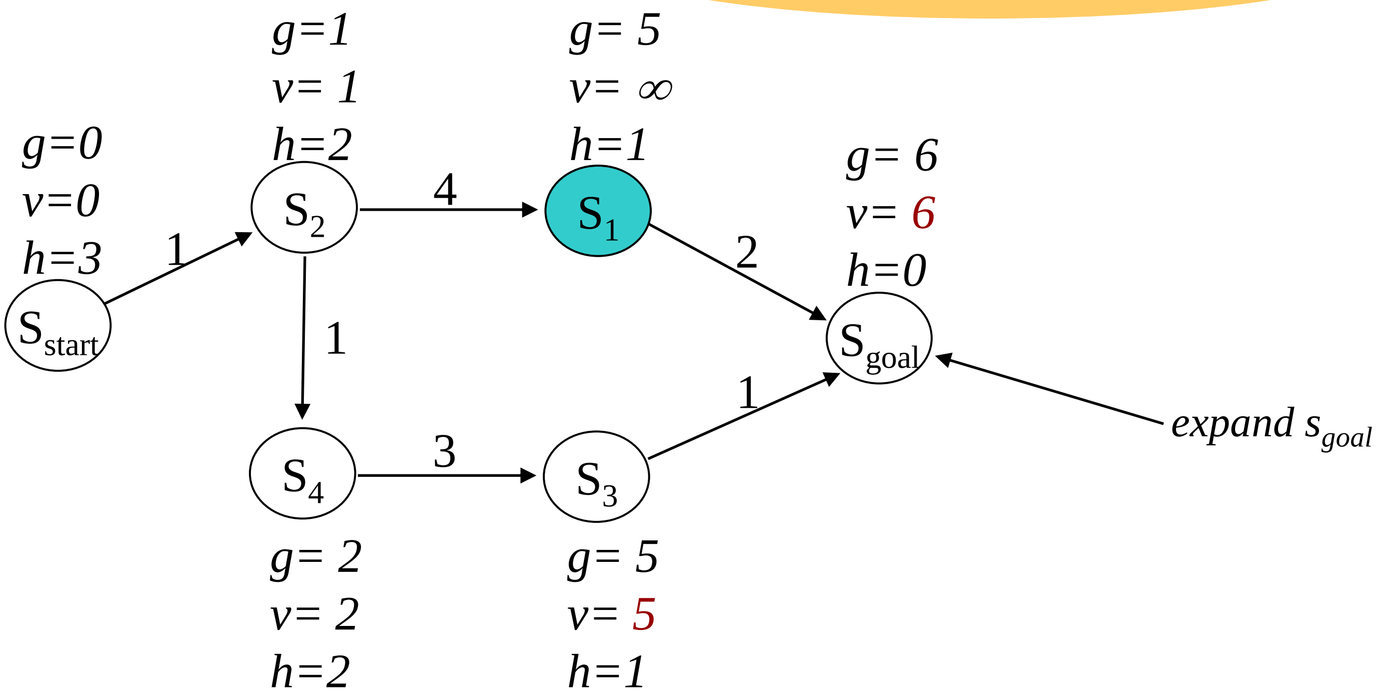


A* with Reuse of State Values

- Edge cost increases may introduce underconsistent states (v(s) < g(s))
- Fix these by setting *v(s)* = ∞
- Makes *s* overconsistent or consistent *v(s)* ≥ *g(s)*
- Propagate the changes

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

no more underconsistent states!

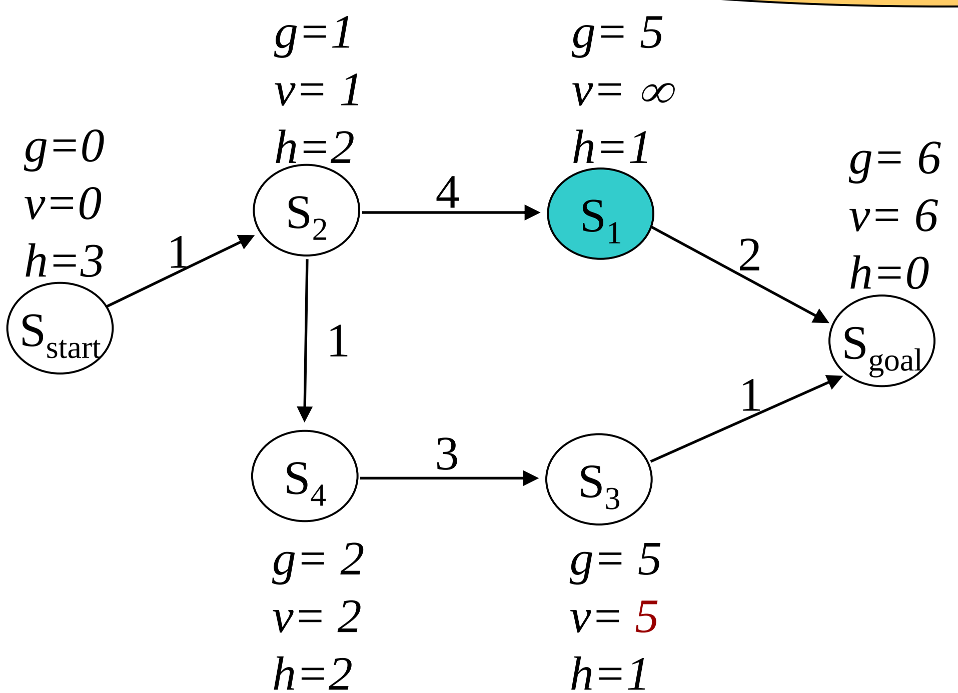


A* with Reuse of State Values

- Edge cost increases may introduce underconsistent states (v(s) < g(s))
- Fix these by setting *v(s)* = ∞
- Makes *s* overconsistent or consistent
- Propagate the changes

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

we can backtrack an optimal path
(start at s_{goal}, proceed to pred that minimizes g+c)

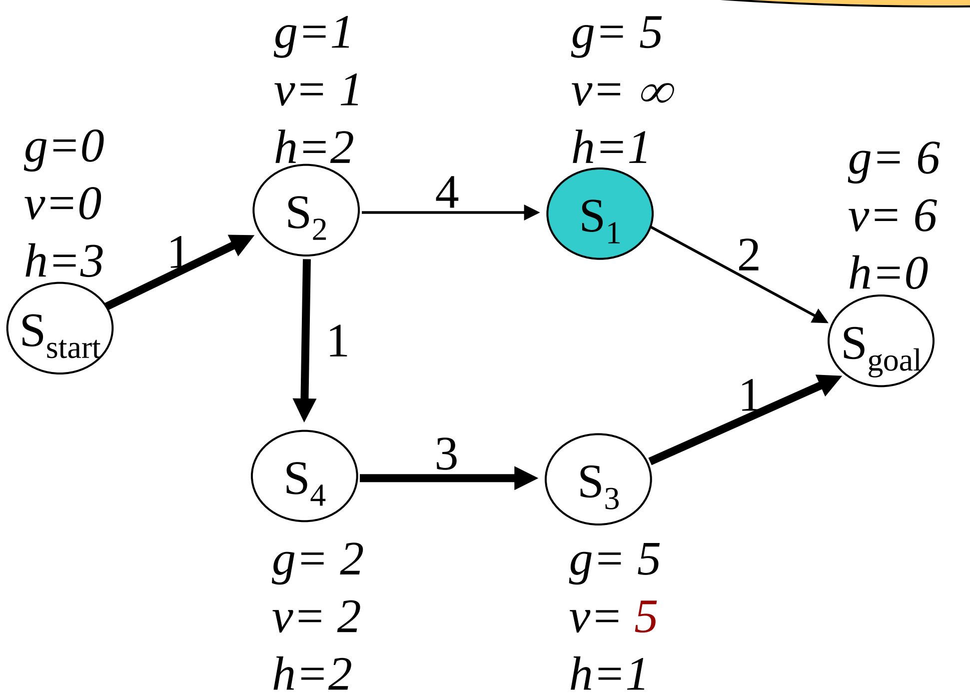


A* with Reuse of State Values

- Edge cost increases may introduce underconsistent states (v(s) < g(s))
- Fix these by setting *v(s)* = ∞
- Makes *s* overconsistent or consistent
- Propagate the changes

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

we can backtrack an optimal path
(start at s_{goal}, proceed to pred that minimizes g+c)



D* Lite

- Optimal re-planning algorithm
- Simpler and with nicer theoretical properties version of D*

```
until goal is reached
    ComputePathwithReuse();   //modified to fix underconsistent states
    publish optimal 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;
```

D* Lite

- Optimal re-planning algorithm
- Simpler and with nicer theoretical properties version of D*

until goal is reached
    ComputePathwithReuse();   *//modified to fix underconsistent states*
    publish optimal 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;

Important detail! search is done backwards:
search starts at s_{goal}, and searhes towards s_{start} with all edges reversed

This way, root of the search tree remains the same and g-values are more likely to remain the same in between two calls to ComputePathwithReuse

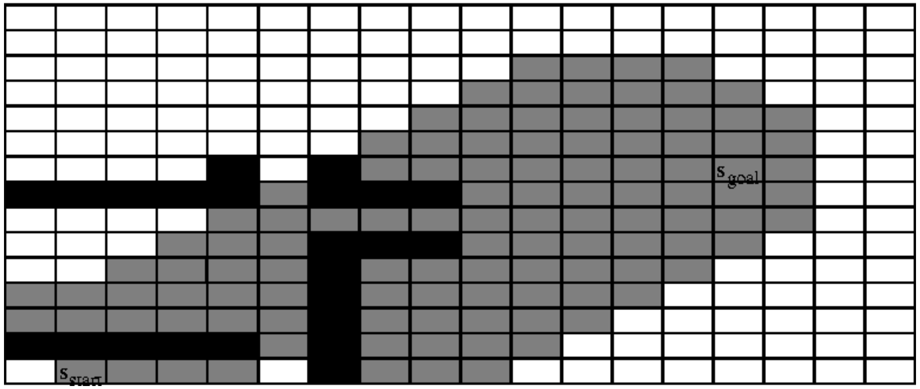
why?

why care?

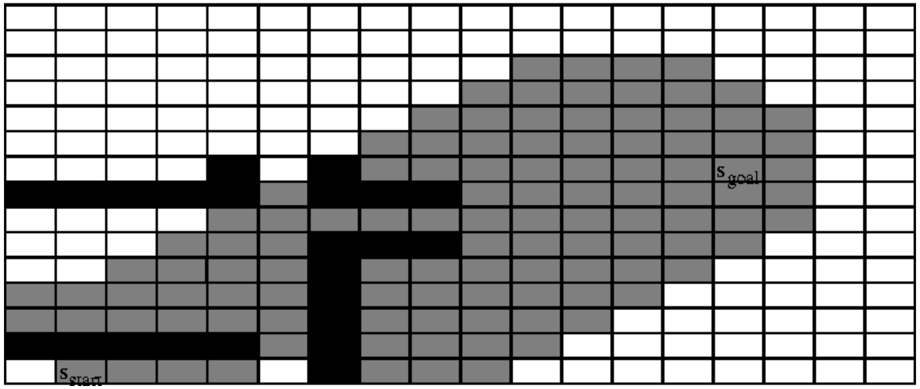
D* Lite

- D* & D* Lite vs. A*

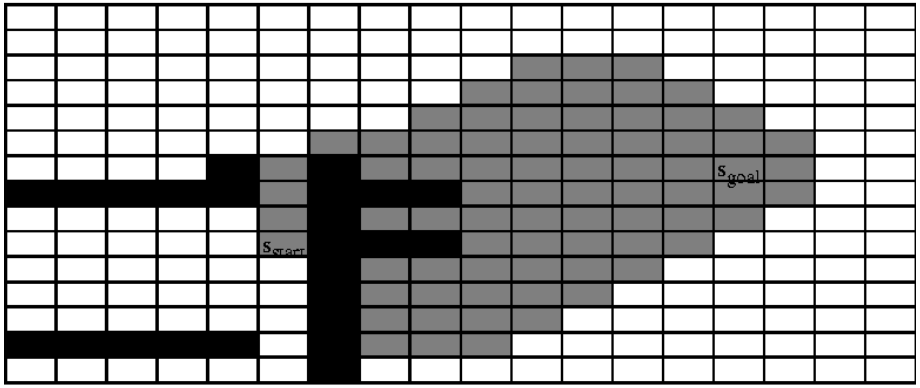
initial A* search



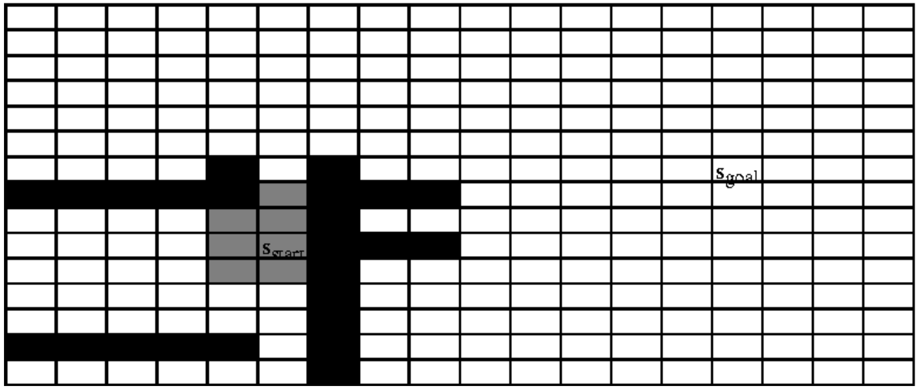
initial D* Lite search



second A* search



second D* Lite search



Anytime and Incremental Planning

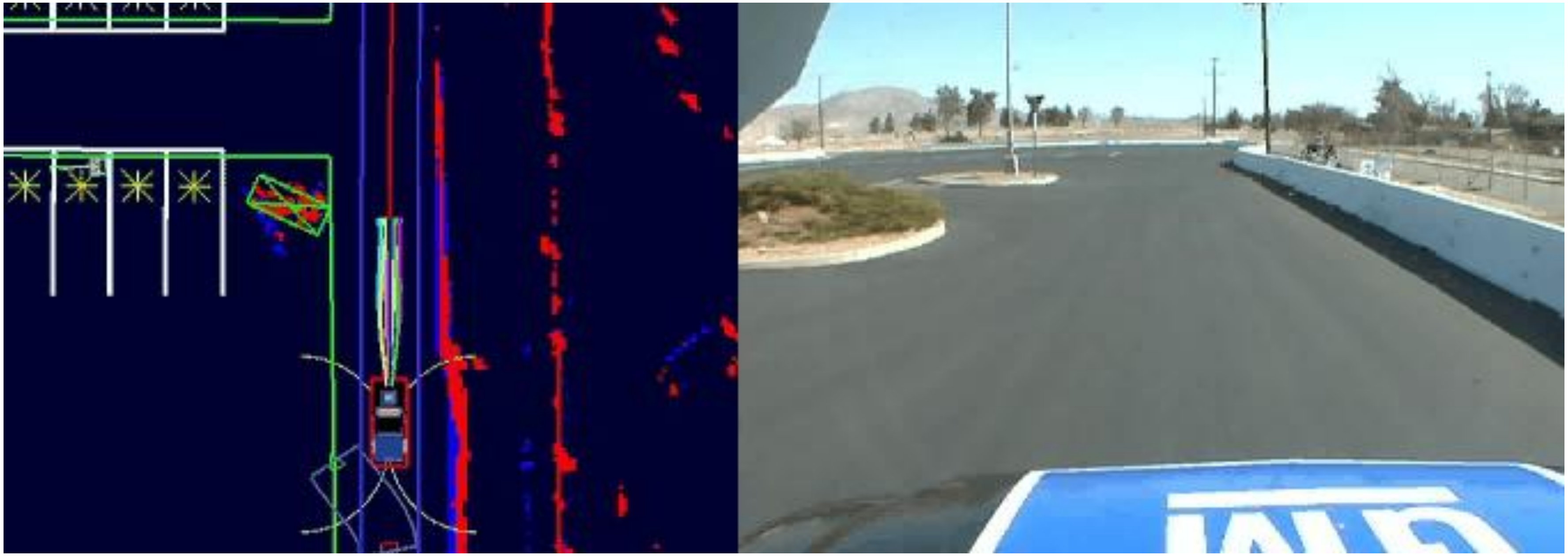
- Anytime D*:
  - decrease ε and update edge costs at the same time
  - re-compute a path by reusing previous state-values

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;

What for?

Anytime and Incremental Planning

- Anytime D* in Urban Challenge



Tartanracing, CMU

Other Uses of Incremental A*

- Whenever planning is a repeated process:
  - improving a solution (e.g., in anytime planning)
  - re-planning in dynamic and previously unknown environments
  - adaptive discretization
  - many other planning problems can be solved via iterative planning