16-782
Planning & Decision-making in Robotics

*Interleaving Planning & Execution: Anytime Incremental A**

Maxim Likhachev

Robotics Institute

Carnegie Mellon University

Planning during Execution

- Planning is a repeated process!

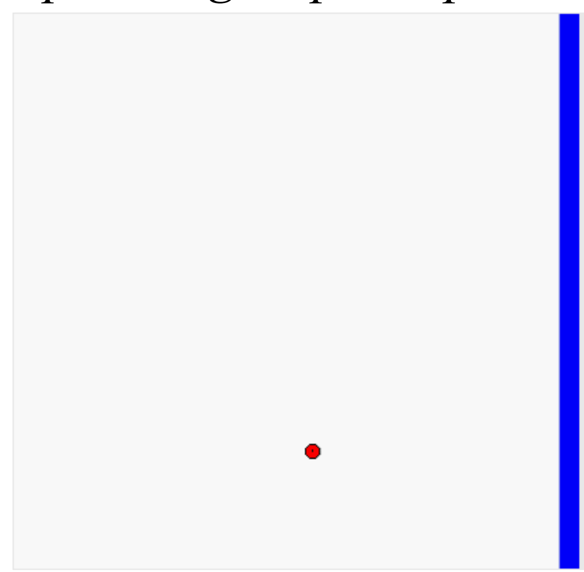
Reasons?

Planning during Execution

- Planning is a repeated process!
  - partially-known environments
  - dynamic environments
  - imperfect execution of plans
  - imprecise localization

ATRV navigating initially-unknown environment

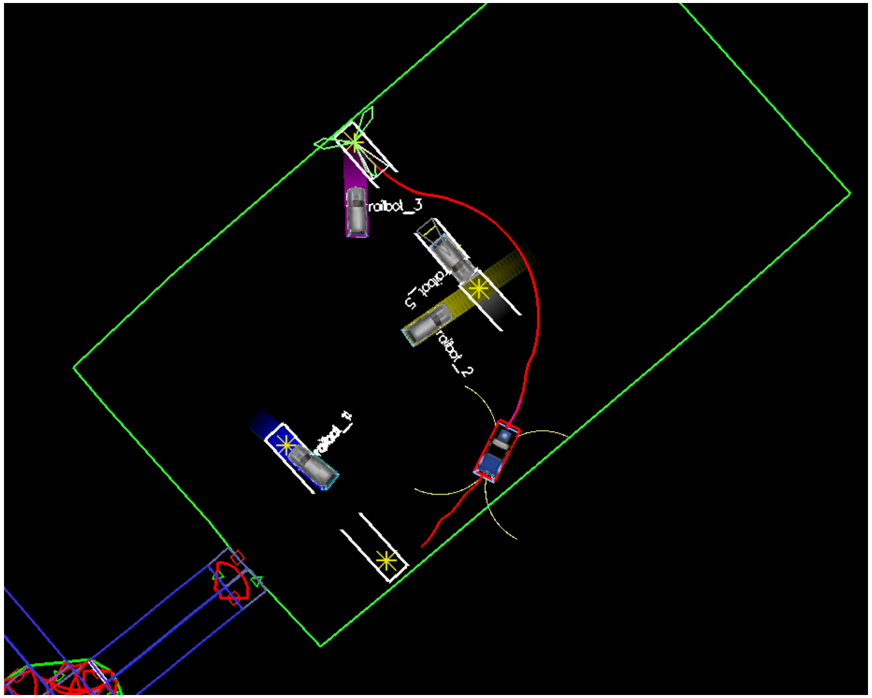
planning map and path



Planning during Execution

- Planning is a <u>repeated</u> process!
  - partially-known environments
  - dynamic environments
  - imperfect execution of plans
  - imprecise localization

planning in dynamic environments



Planning during Execution

- Planning is a repeated process!
  - partially-known environments
  - dynamic environments
  - imperfect execution of plans
  - imprecise localization

- Need to be able to re-plan fast!

- Several methodologies to achieve this:
  - anytime heuristic search: return the best plan possible within T msecs
  - incremental heuristic search: speed up search by reusing previous efforts
  - real-time heuristic search: plan few steps towards the goal and re-plan later

Planning during Execution

- Planning is a <u>repeated</u> process!
  - partially-known environments
  - dynamic environments
  - imperfect execution of plans
  - imprecise localization

- Need to be able to re-plan fast!

- Several methodologies to achieve this:

  this class

  - anytime heuristic search: return the best plan possible within T msecs
  - incremental heuristic search: speed up search by reusing previous efforts
  - real-time heuristic search: plan few steps towards the goal and re-plan later

  next class

Planning during Execution

- Planning is a <u>repeated</u> process!
  - partially-known environments
  - dynamic environments
  - imperfect execution of plans
  - imprecise localization

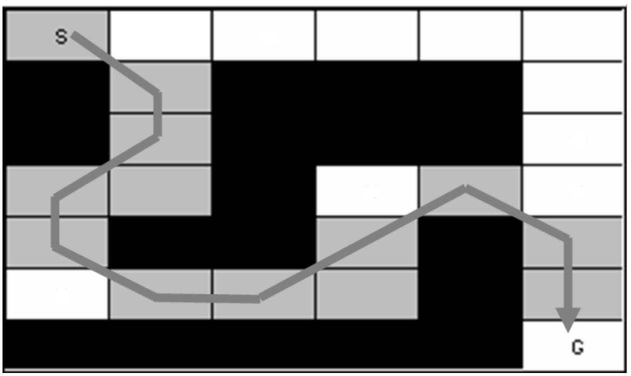
- Need to be able to re-plan fast!

- Several methodologies to achieve this:
  - anytime heuristic search: return the best plan possible within T msecs
  - incremental heuristic search: speed up search by reusing previous efforts
  - real-time heuristic search: plan few steps towards the goal and re-plan later

Anytime Heuristic Search: Straw Man Approach

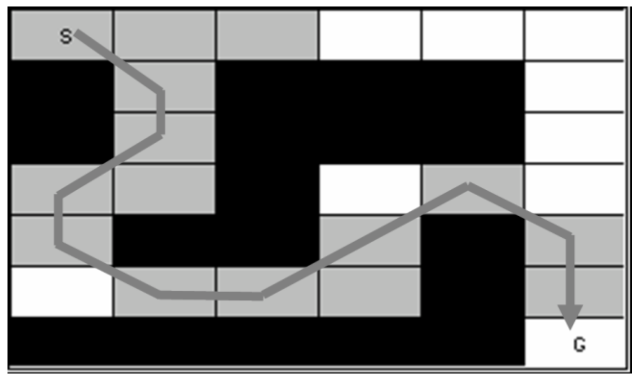
- Constructing anytime search based on weighted A*:
  - find the best path possible given some amount of time for planning
  - do it by running a series of weighted A* searches with decreasing *ε*:

ε =2.5



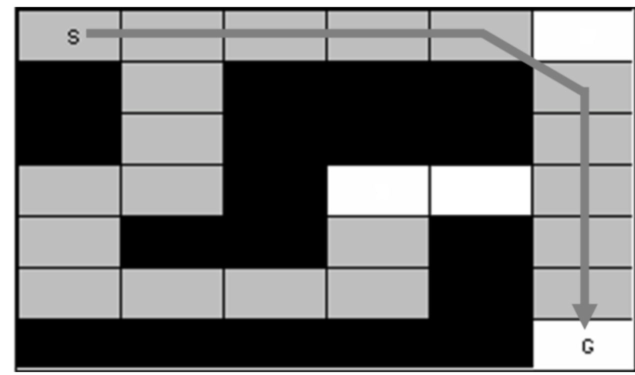
13 expansions
solution=11 moves

ε =1.5



15 expansions
solution=11 moves

ε =1.0

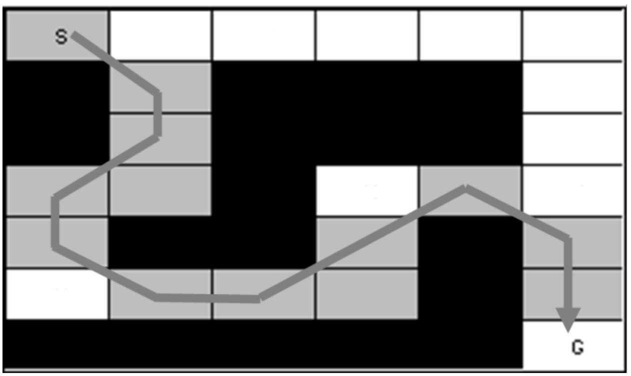


20 expansions
solution=10 moves

Anytime Heuristic Search: Straw Man Approach

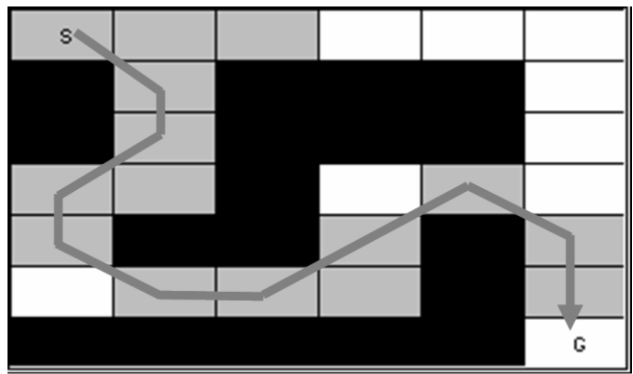
- Constructing anytime search based on weighted A*:
  - find the best path possible given some amount of time for planning
  - do it by running a series of weighted A* searches with decreasing ε:

$\varepsilon = 2.5$



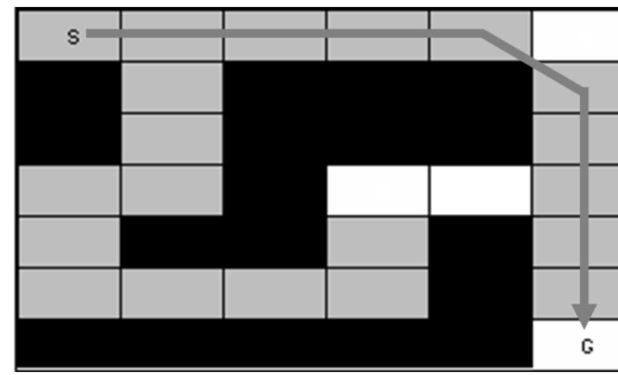
13 expansions
solution=11 moves

$\varepsilon = 1.5$



15 expansions
solution=11 moves

$\varepsilon = 1.0$



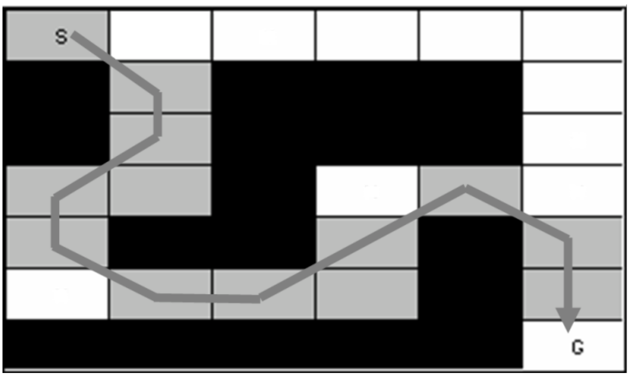
20 expansions
solution=10 moves

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

Anytime Heuristic Search: Straw Man Approach

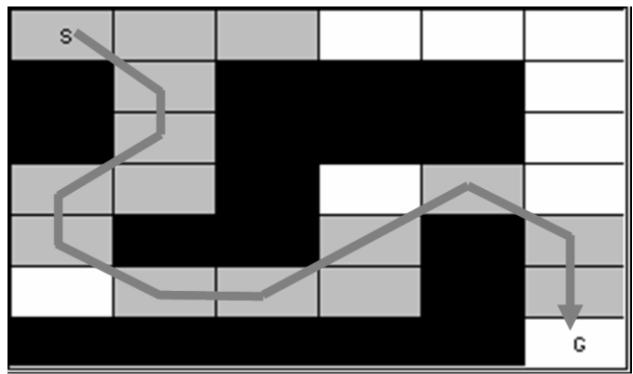
- Constructing anytime search based on weighted A*:
  - find the best path possible given some amount of time for planning
  - do it by running a series of weighted A* searches with decreasing *ε*:

ε =2.5



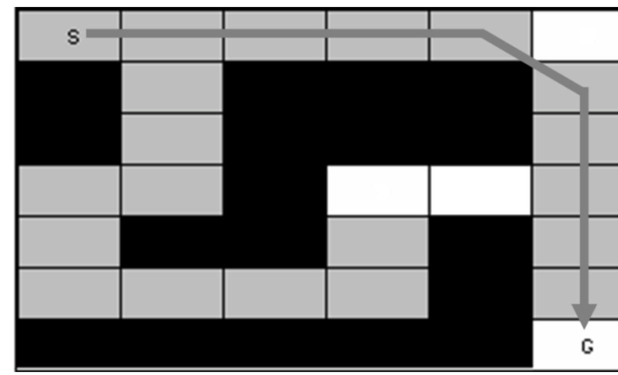
13 expansions
solution=11 moves

ε =1.5



15 expansions
solution=11 moves

ε =1.0



20 expansions
solution=10 moves

- ARA* [Likhachev et al., '04]
  - efficient version of above that reuses state values between iterations

A* with Reuse of State Values

- Alternative view of A*

all *v*-values initially are infinite;

ComputePath function

while(s_{goal} is not expanded AND $OPEN \neq 0$)

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 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* with Reuse of State Values

- Alternative view of A*

all *v*-values initially are infinite;

ComputePath function

while(s_{goal} is not expanded AND $OPEN \neq 0$)

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 such that s' not in $CLOSED$

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

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

insert s' into $OPEN$;

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

A* with Reuse of State Values

- Alternative view of A*

all *v*-values initially are infinite;

ComputePath function

while(s_{goal} is not expanded AND $OPEN \neq 0$)

  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 such that s' not in $CLOSED$

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

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

      insert s' into $OPEN$;

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

A* with Reuse of State Values

- Alternative view of A*

all v-values initially are infinite;

ComputePath function

while(s_{goal} is not expanded AND $OPEN \neq 0$)

  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 such that s' not in $CLOSED$

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

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

      insert s' into $OPEN$;

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

Why?

A* with Reuse of State Values

- Alternative view of A*

all *v*-values initially are infinite;

ComputePath function

while(s_{goal} is not expanded AND $OPEN \neq 0$)

  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* 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

A* with Reuse of State Values

- Alternative view of A*

all *v*-values initially are infinite;

ComputePath function

while(s_{goal} is not expanded AND $OPEN \neq 0$)

  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* 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

Why?

A* with Reuse of State Values

- Alternative view of A*

all *v*-values initially are infinite;

ComputePath function

while(s_{goal} is not expanded AND $OPEN \neq 0$)

  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* 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 $v(s) = g(s)$
- 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 such that s' not in *CLOSED*

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

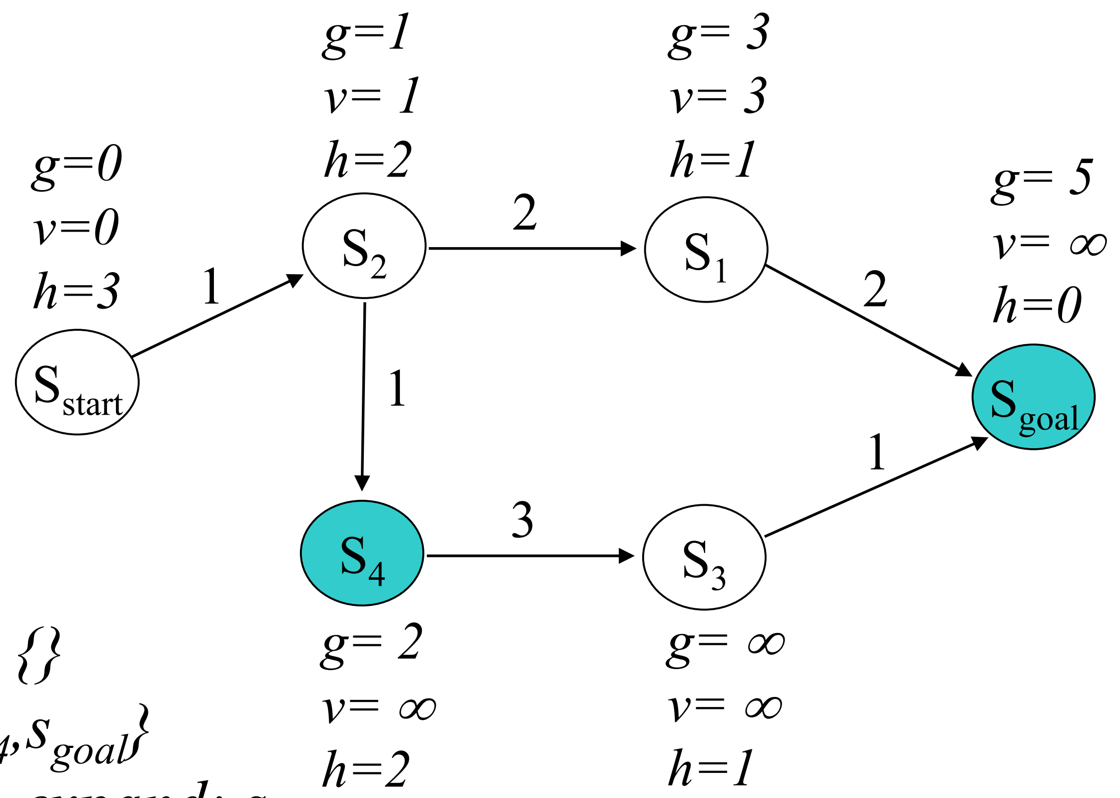
      $g(s') = g(s) + c(s,s')$;

      insert s' into *OPEN*;

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

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

A* with Reuse of State Values



CLOSED = {}

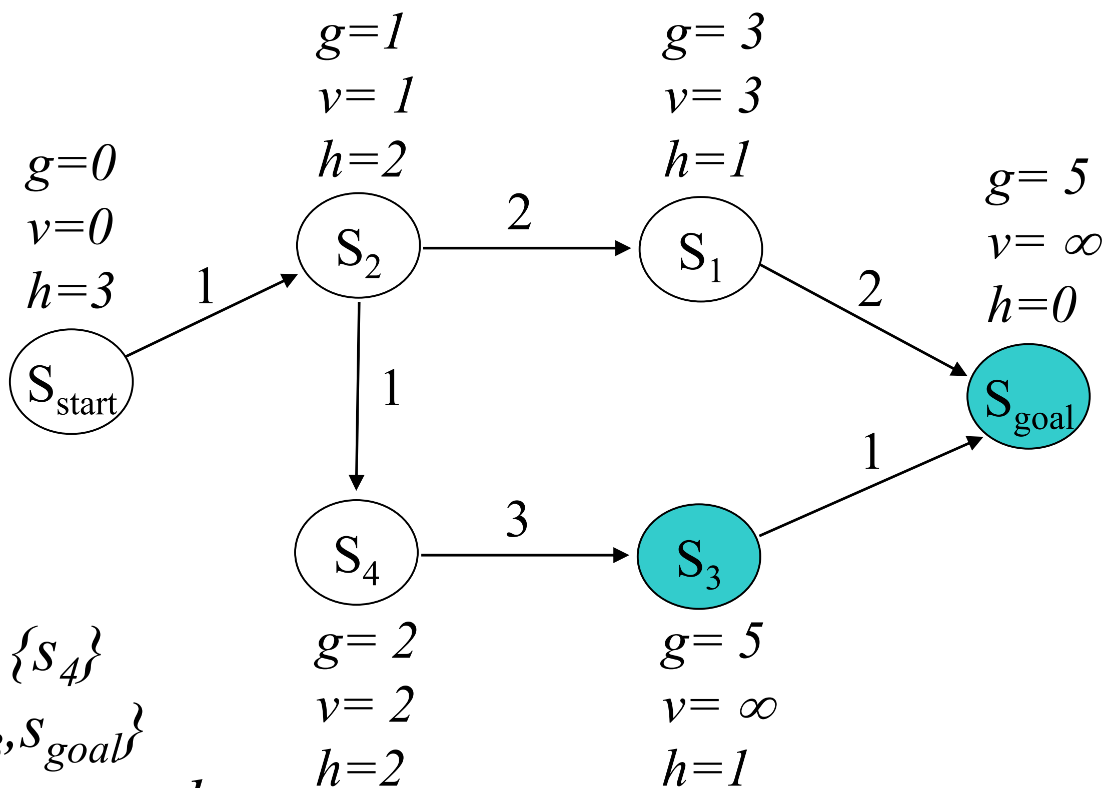
OPEN = {s_4, s_{goal}}

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

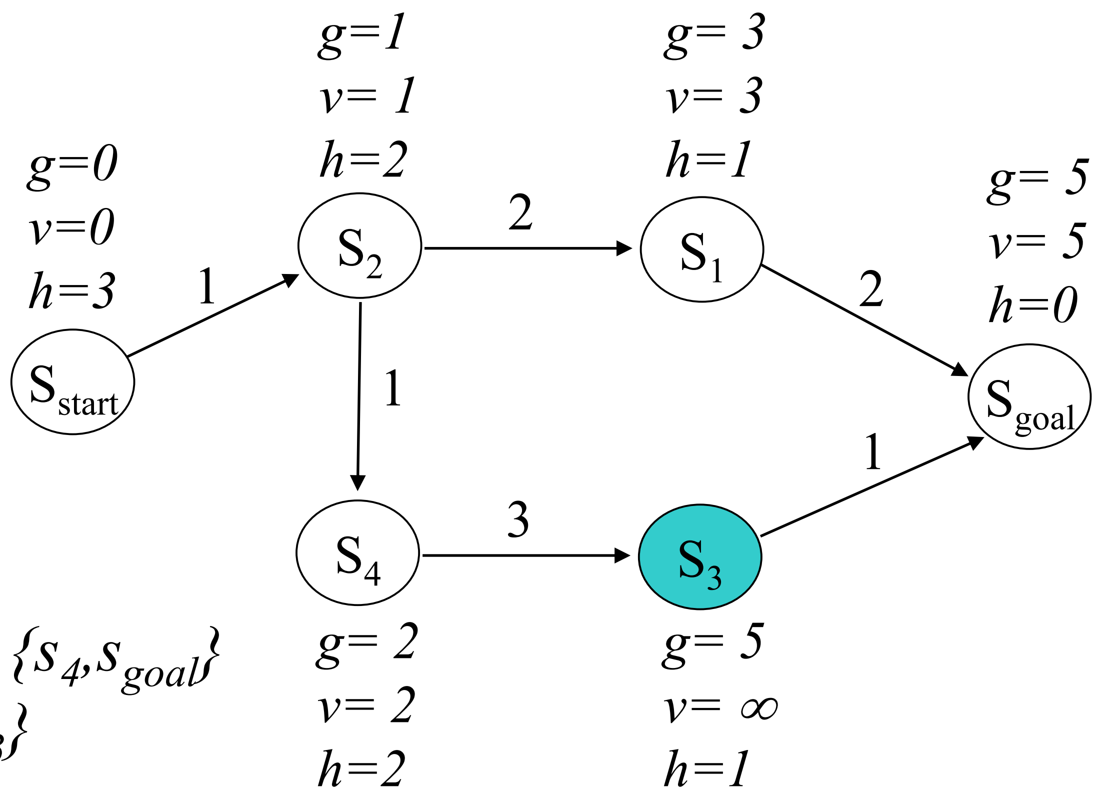


CLOSED = $\{s_4\}$

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

next state to expand: s_{goal}

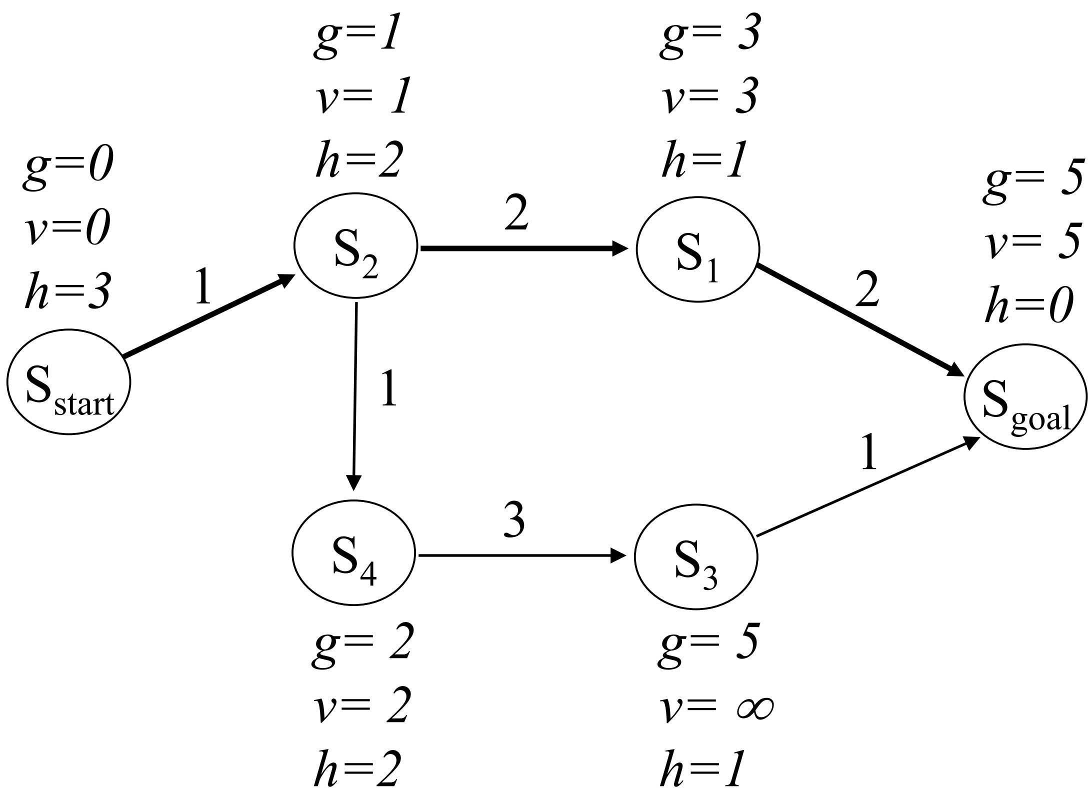
A* with Reuse of State Values



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 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*;

*the exact same thing as with A**

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**

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

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*;

Does OPEN contain ALL overconsistent states (i.e., states s' whose $v(s') > g(s')$)?

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;

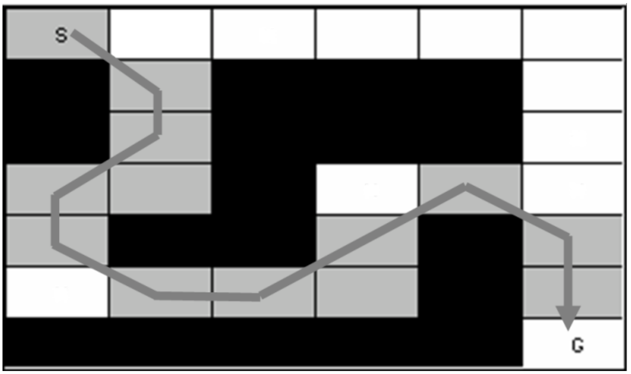
  initialize $OPEN = OPEN \cup INCONS$;

all overconsistent states (exactly what we need!)

ARA*

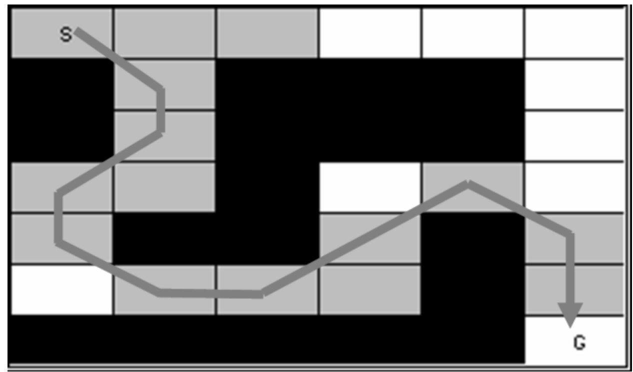
- **A series of weighted A* searches**

ε =2.5



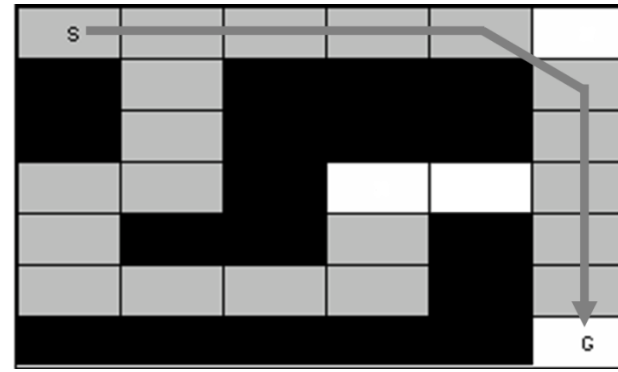
13 expansions
solution=11 moves

ε =1.5



15 expansions
solution=11 moves

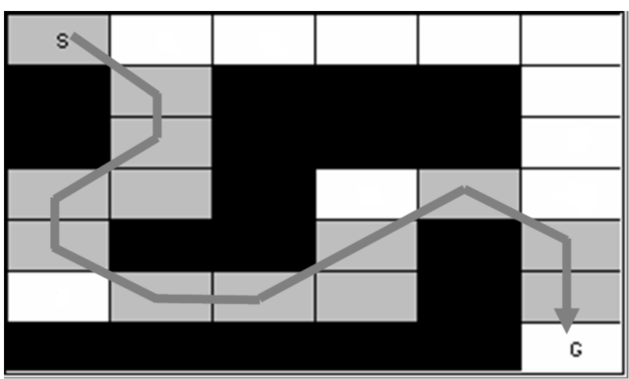
ε =1.0



20 expansions
solution=10 moves

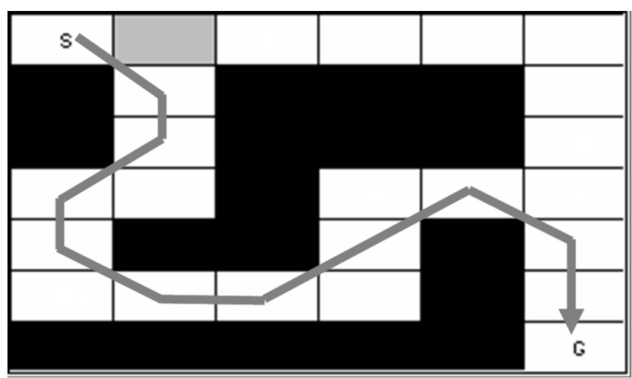
- **ARA***

ε =2.5



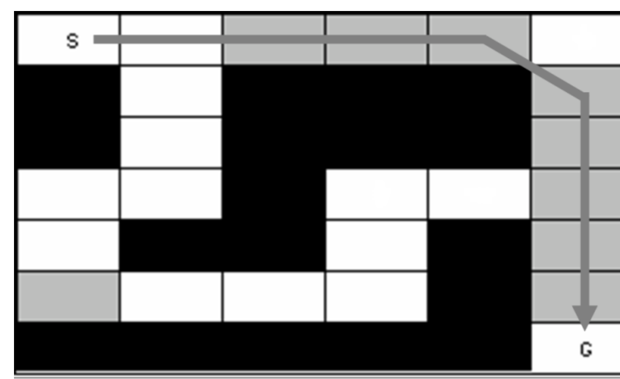
13 expansions
solution=11 moves

ε =1.5



1 expansion
solution=11 moves

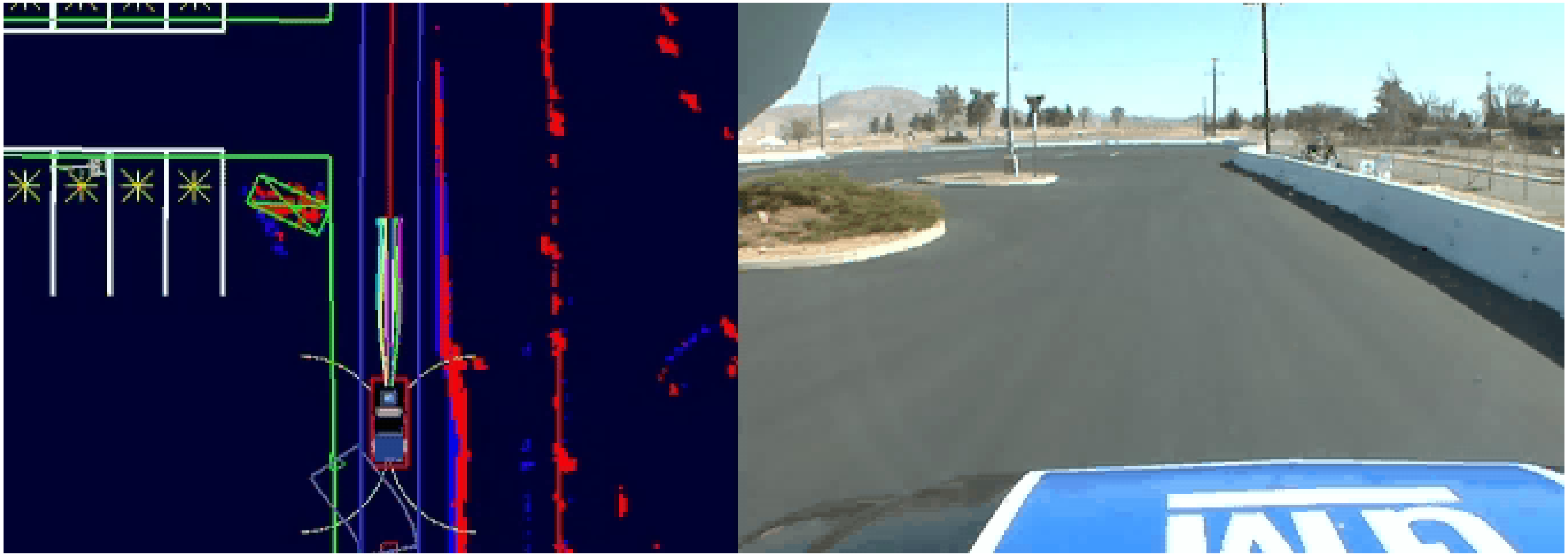
ε =1.0



9 expansions
solution=10 moves

Anytime Heuristic Search in Action

- Anytime D* during Urban Challenge race



Planning during Execution

- Planning is a repeated process!
  - partially-known environments
  - dynamic environments
  - imperfect execution of plans
  - imprecise localization

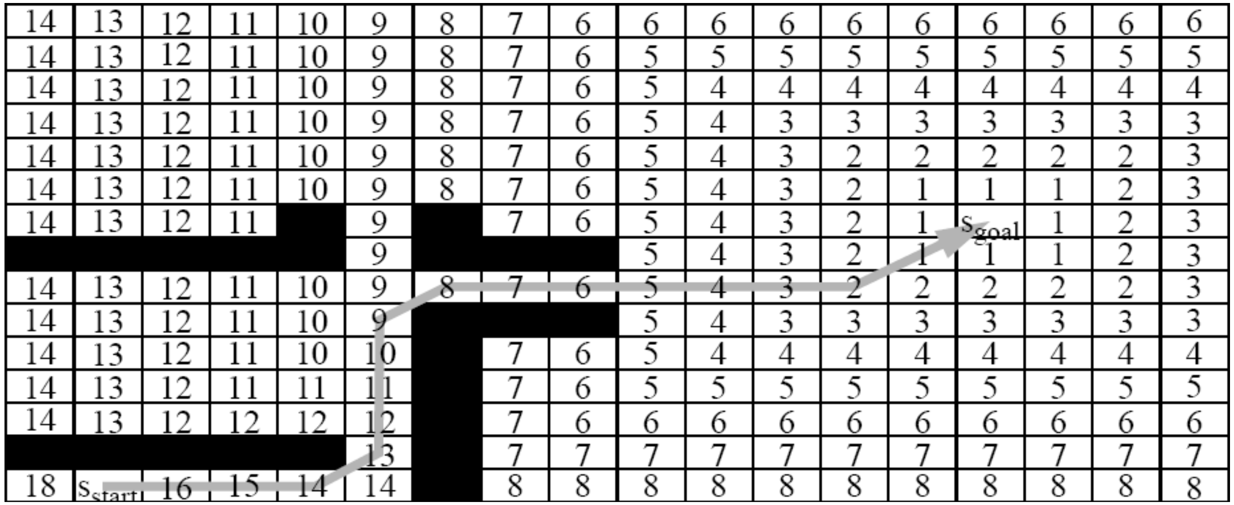
- Need to be able to re-plan fast!

- Several methodologies to achieve this:
  - anytime heuristic search: return the best plan possible within T msecs
  - incremental heuristic search: speed up search by reusing previous efforts
  - real-time heuristic search: plan few steps towards the goal and re-plan later

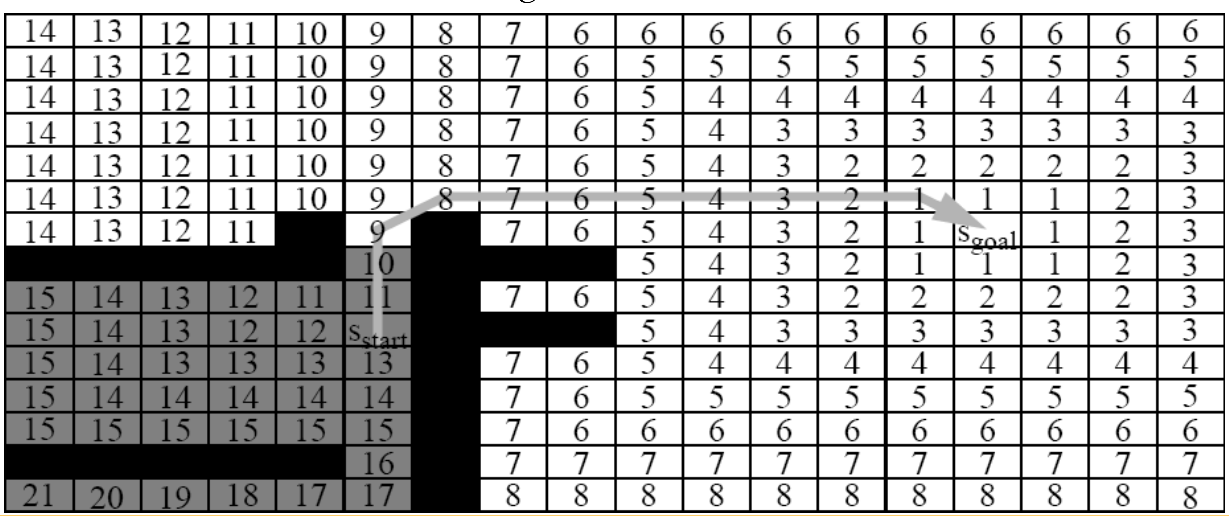
Incremental Heuristic Search

- 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

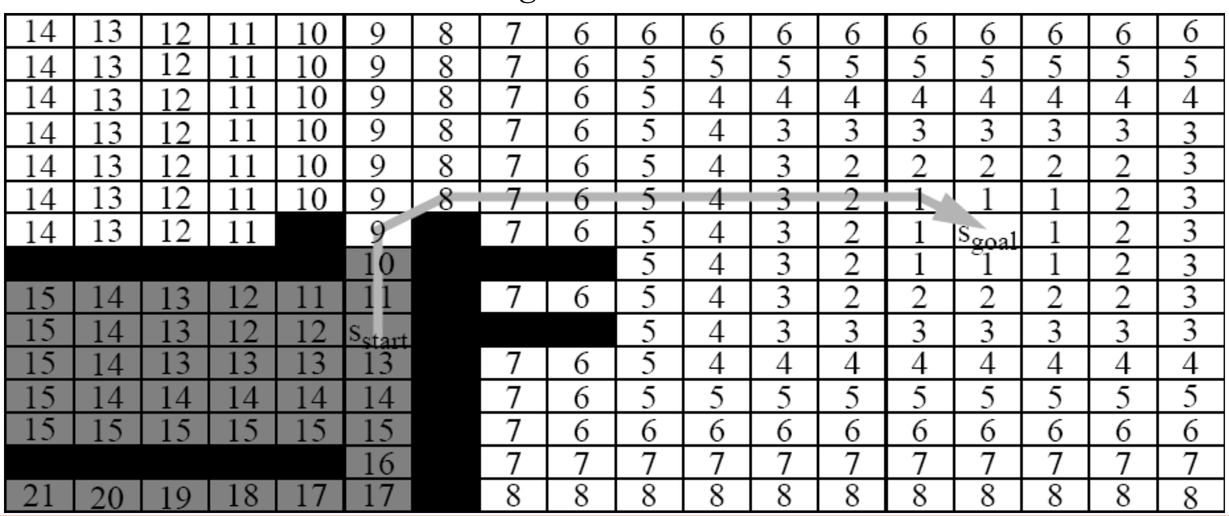


Incremental Heuristic Search

- 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



Incremental Heuristic Search

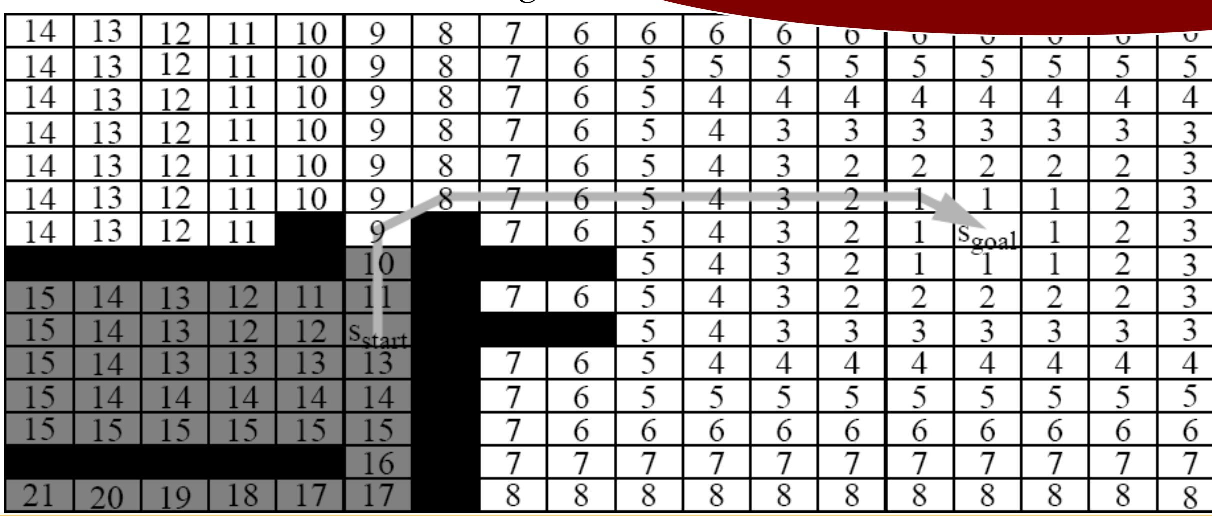
- Reuse state values from previous searches

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**

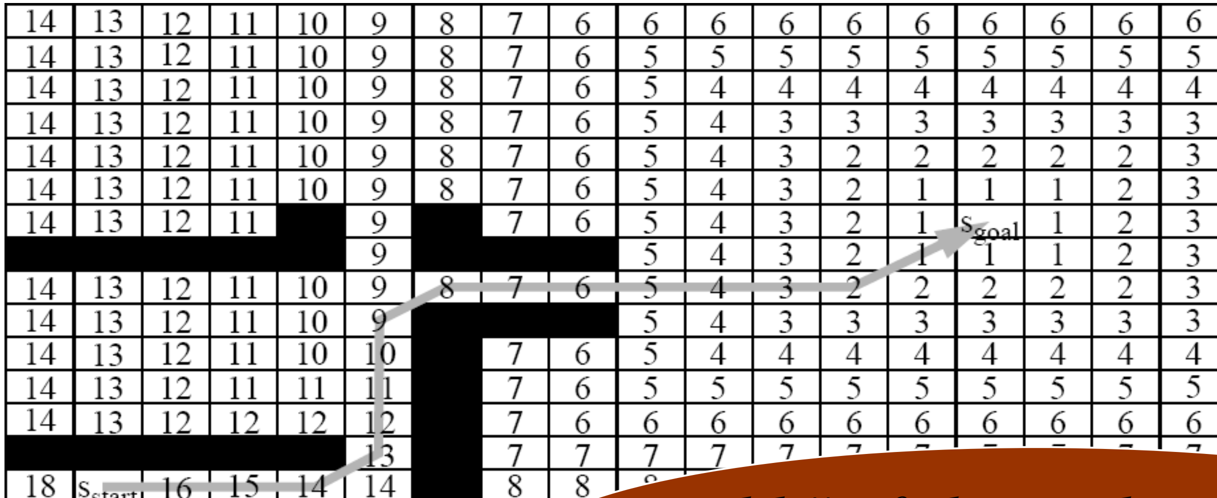
cost of least-cost paths to s_{goal} ...



Incremental Heuristic Search

- Reuse state values from previous searches

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



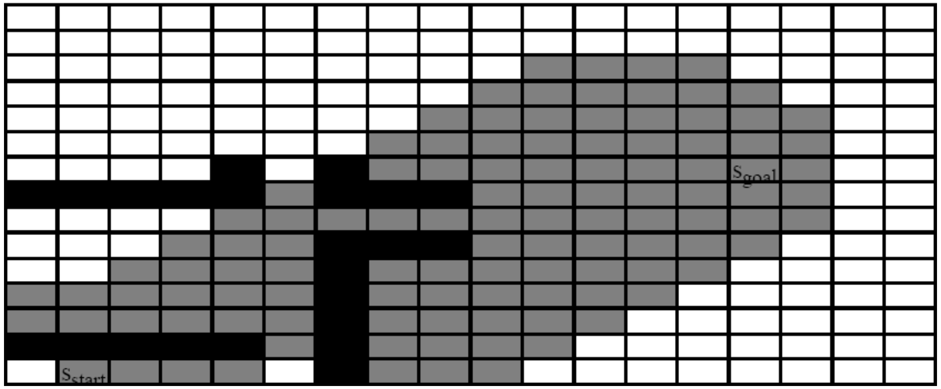
cost of least-cost paths to s

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

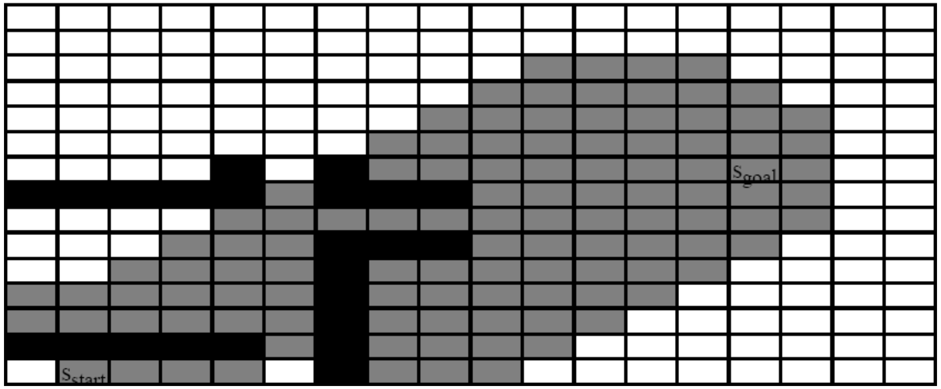
Incremental Heuristic Search

- Reuse state values from previous searches

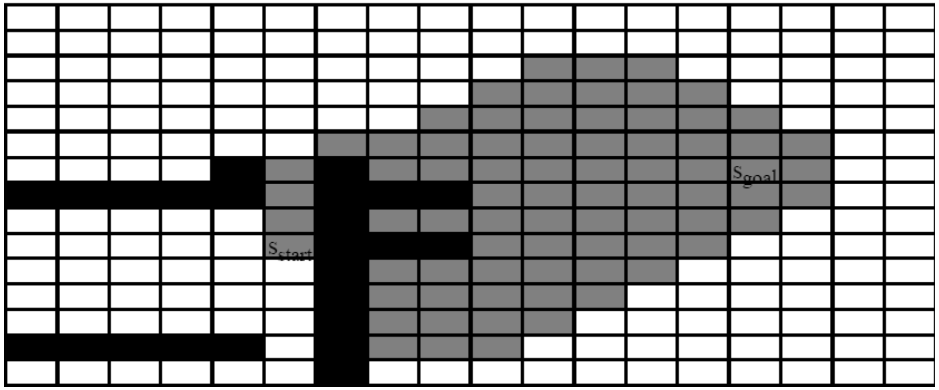
*initial search by backwards A**



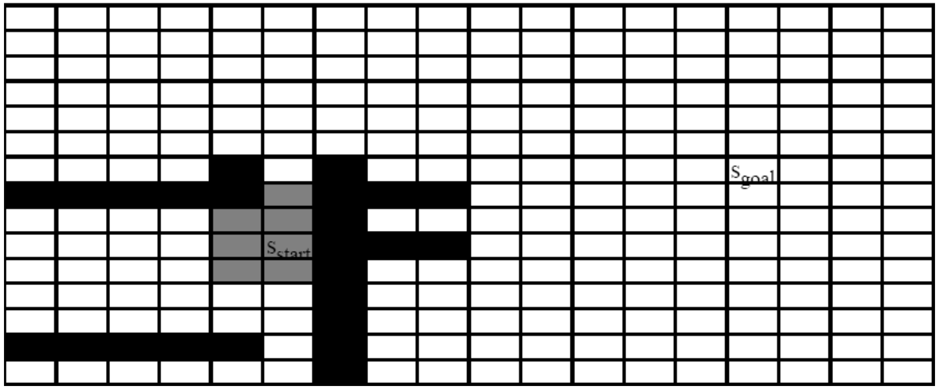
initial search by D Lite*



*second search by backwards A**



second search by D Lite*



Incremental Heuristic Search

- Three general approaches to reusing previous search efforts:
  - Identifying the boundaries of the previously generated search tree that remains to be valid and re-starting the search from it
    - Differential A* [Trovato & Dorst, ‘02], Fringe-Saving A* [Sun & Koenig, ’07], Tree-restoring weighted A* [Gochev et al., ‘14]
  - Fixing the previously generated search tree by re-using as much of it as possible
    - D* [Stentz, ’95], D* Lite [Koenig & Likhachev, ’02], Anytime D* [Likhachev et al., ‘08]
  - Restarting search from scratch but “learning” heuristics values
    - Hierarchical A* [Holte et al., 96], Adaptive A* [Koenig & Likhachev, ‘06], Generalized Adaptive A* [Sun et al., 08]

Incremental Heuristic Search

- Three general approaches to reusing previous search efforts:
  - Identifying the boundaries of the previously generated search tree that remains to be valid and re-starting the search from it
    - Differential A* [Trovato & Dorst, '02], Fringe-Saving A* [Sun & Koenig, '07], Tree-restoring weighted A* [Gochev et al., '14]

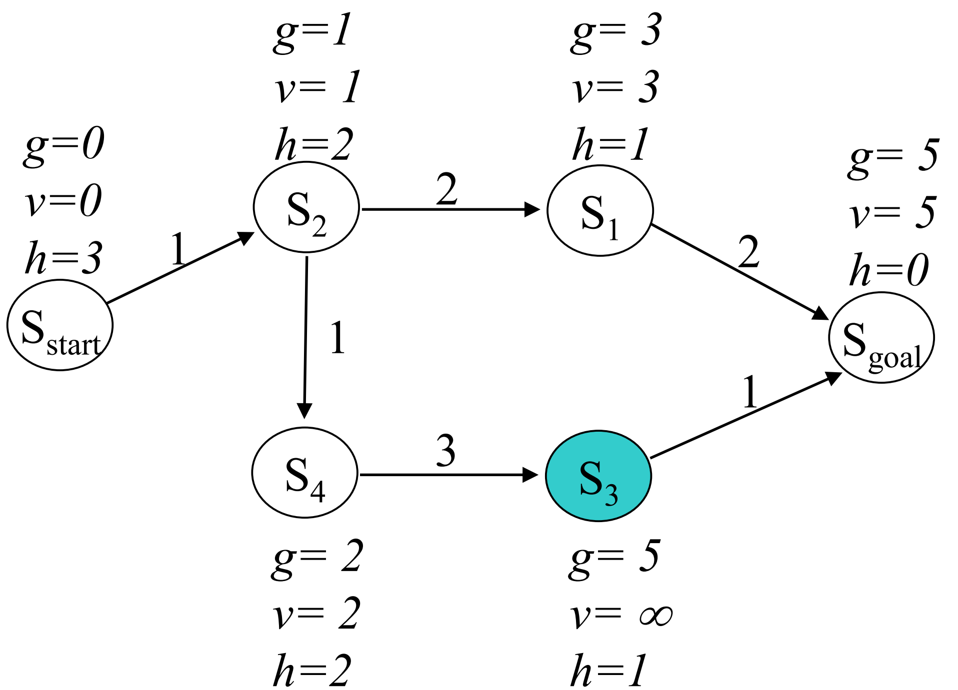
this lecture

  - Fixing the previously generated search tree by re-using as much of it as possible
    - D* [Stentz, '95], D* Lite [Koenig & Likhachev, '02], Anytime D* [Likhachev et al., '08]
  - Restarting search from scratch but "learning" heuristics values
    - Hierarchical A* [Holte et al., 96], Adaptive A* [Koenig & Likhachev, '06], Generalized Adaptive A* [Sun et al., 08]

next lecture

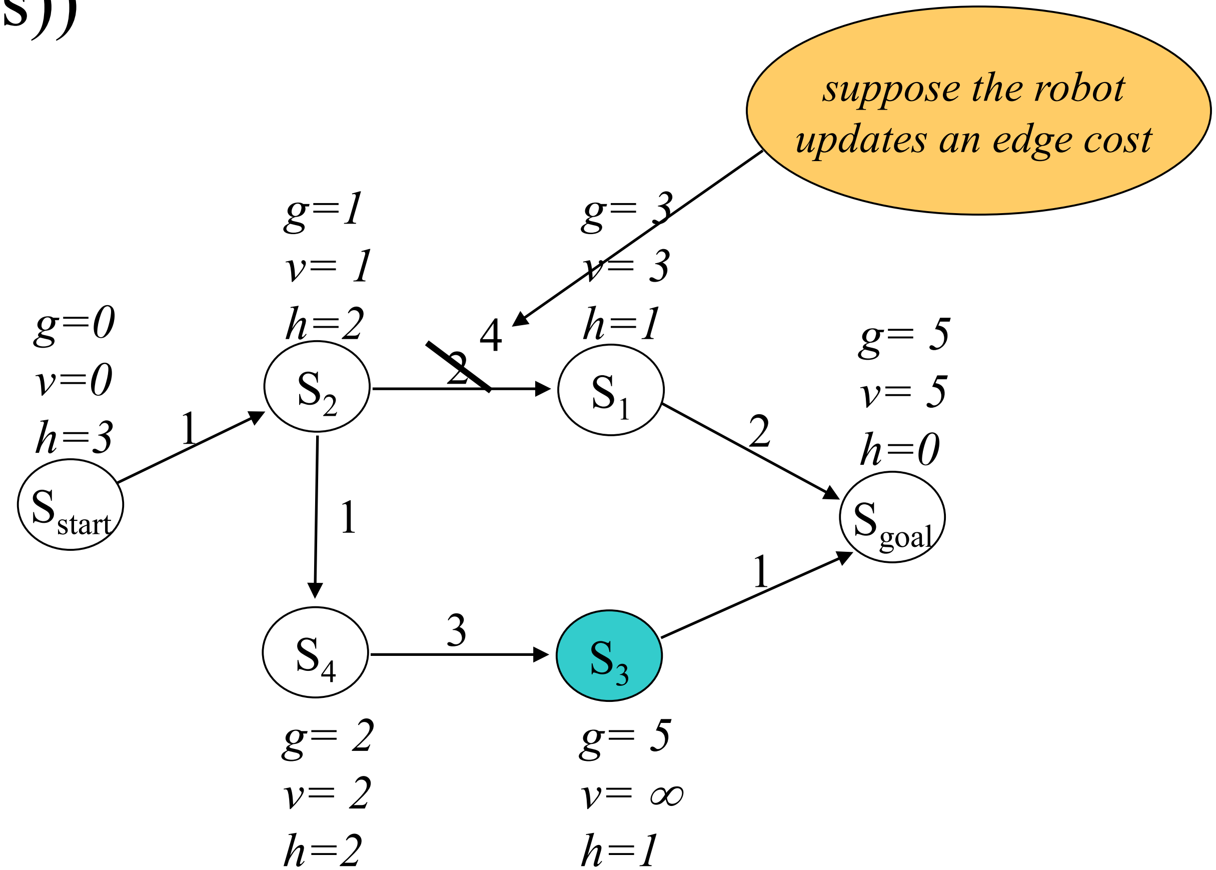
A* with Reuse of State Values

- 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 ($v(s) < g(s)$)

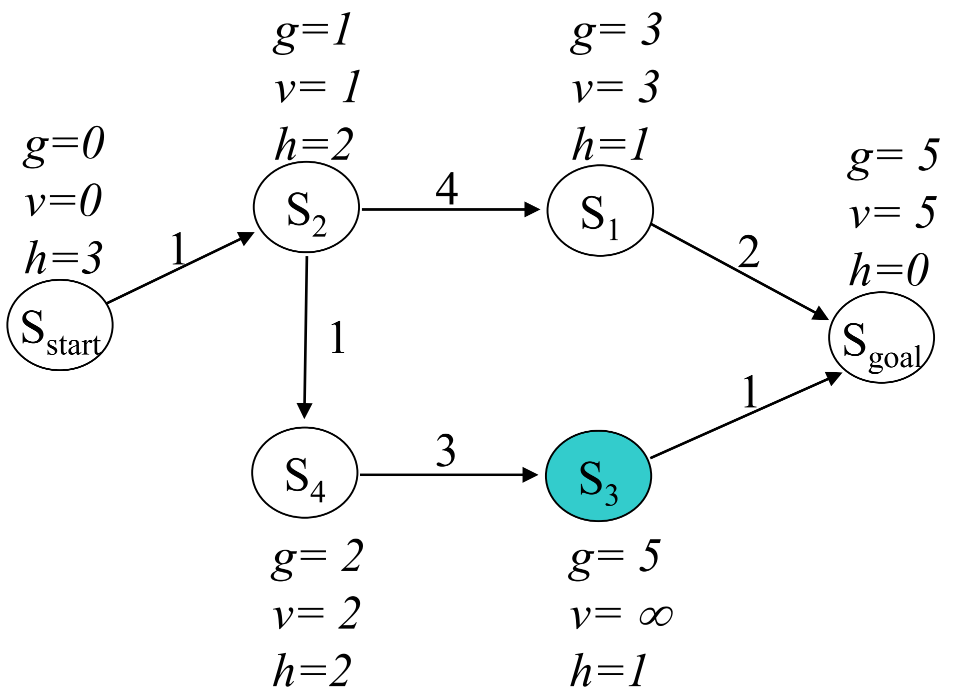


A* with Reuse of State Values

- 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))

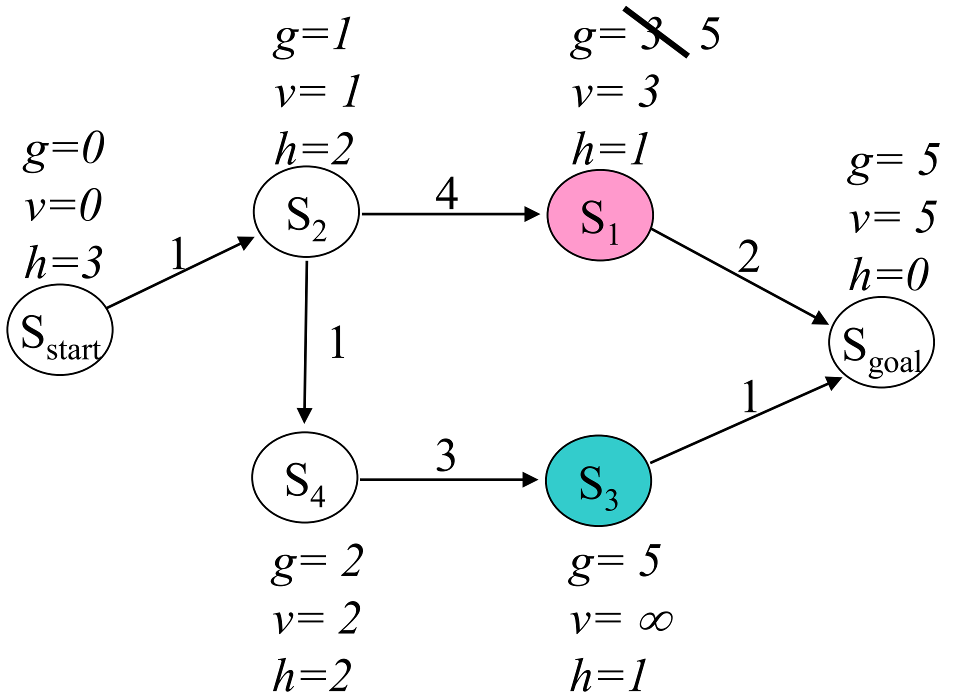
ComputePathwithReuse invariant:
$g(s') = \min_{s'' \in pred(s')} v(s'') + c(s'',s')$

↓

need to update $g(s_1)$

↓

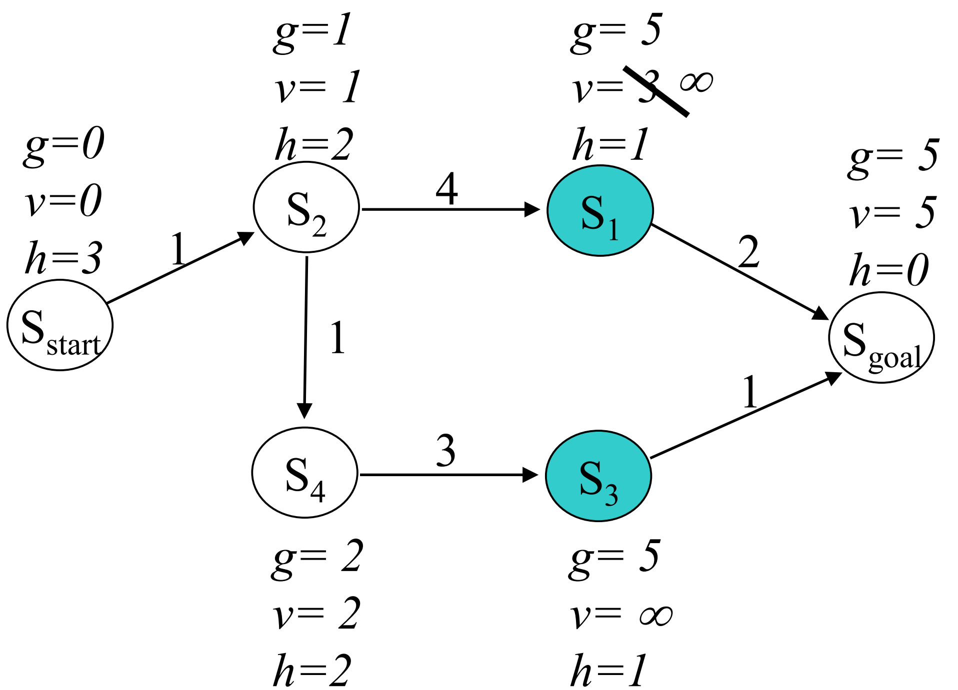
$v(s_1) < g(s_1)$



A* with Reuse of State Values

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

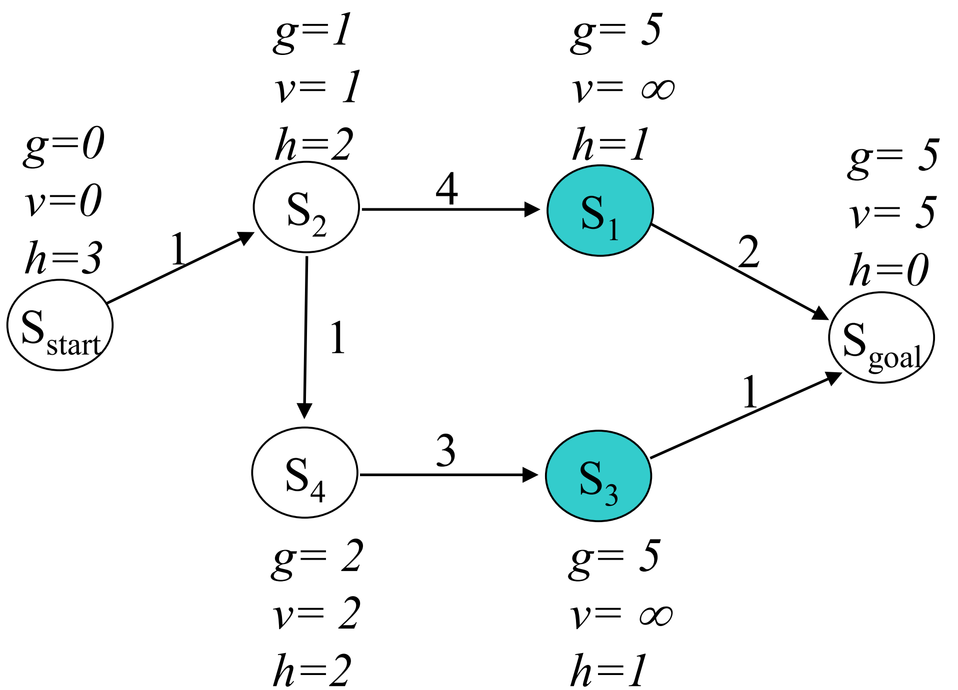
ComputePathwithReuse invariant:
$g(s') = \min_{s'' \in pred(s')} v(s'') + c(s'',s')$



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 $v(s) \geq g(s)$

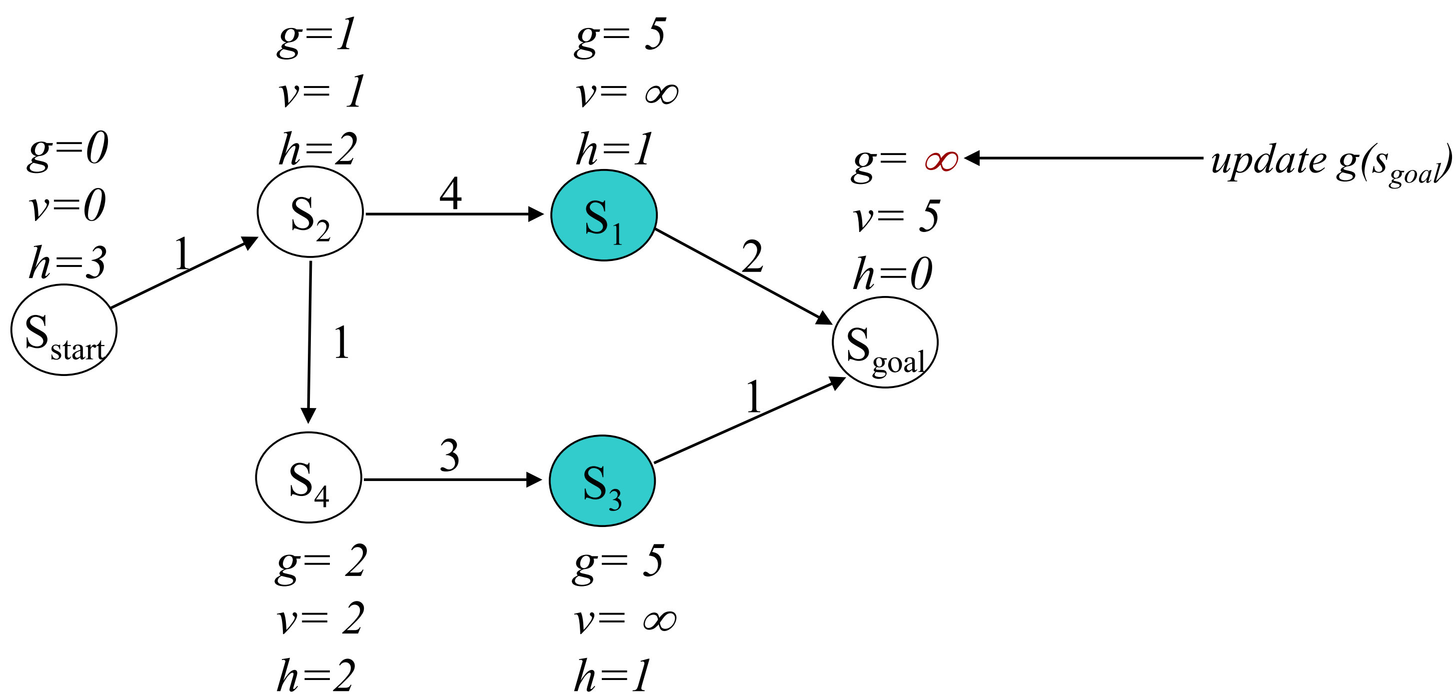
ComputePathwithReuse invariant:
$g(s') = \min_{s'' \in pred(s')} v(s'') + c(s'',s')$



A* with Reuse of State Values

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

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

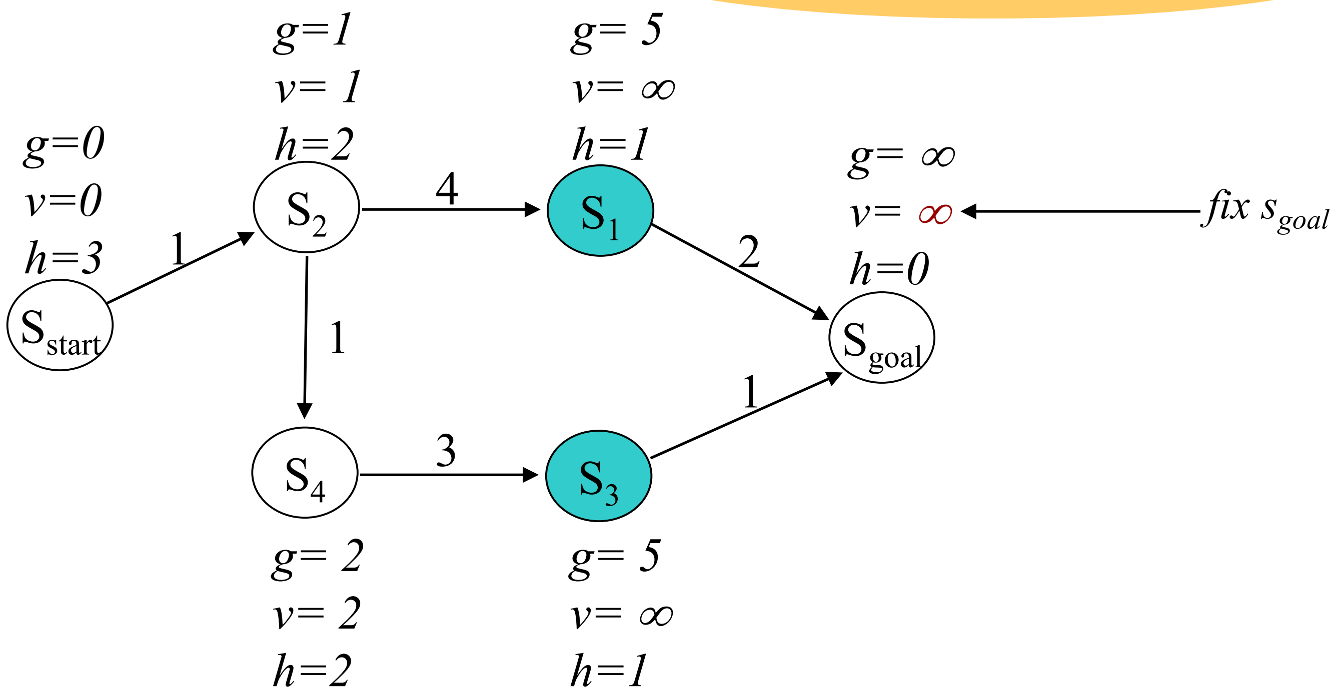


A* with Reuse of State Values

- 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!

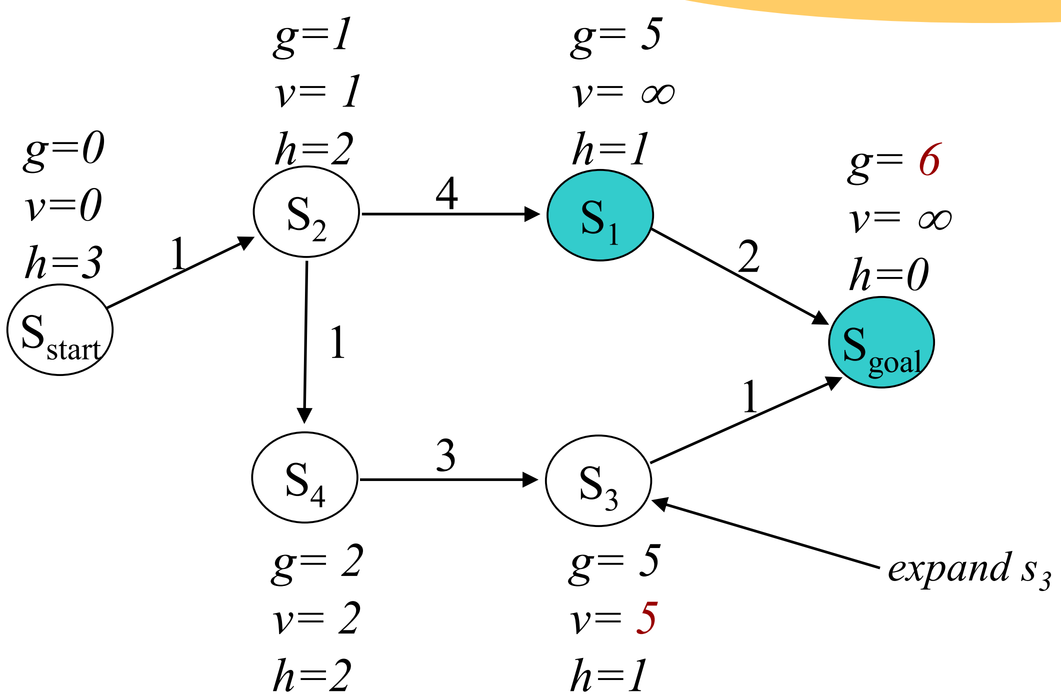


A* with Reuse of State Values

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

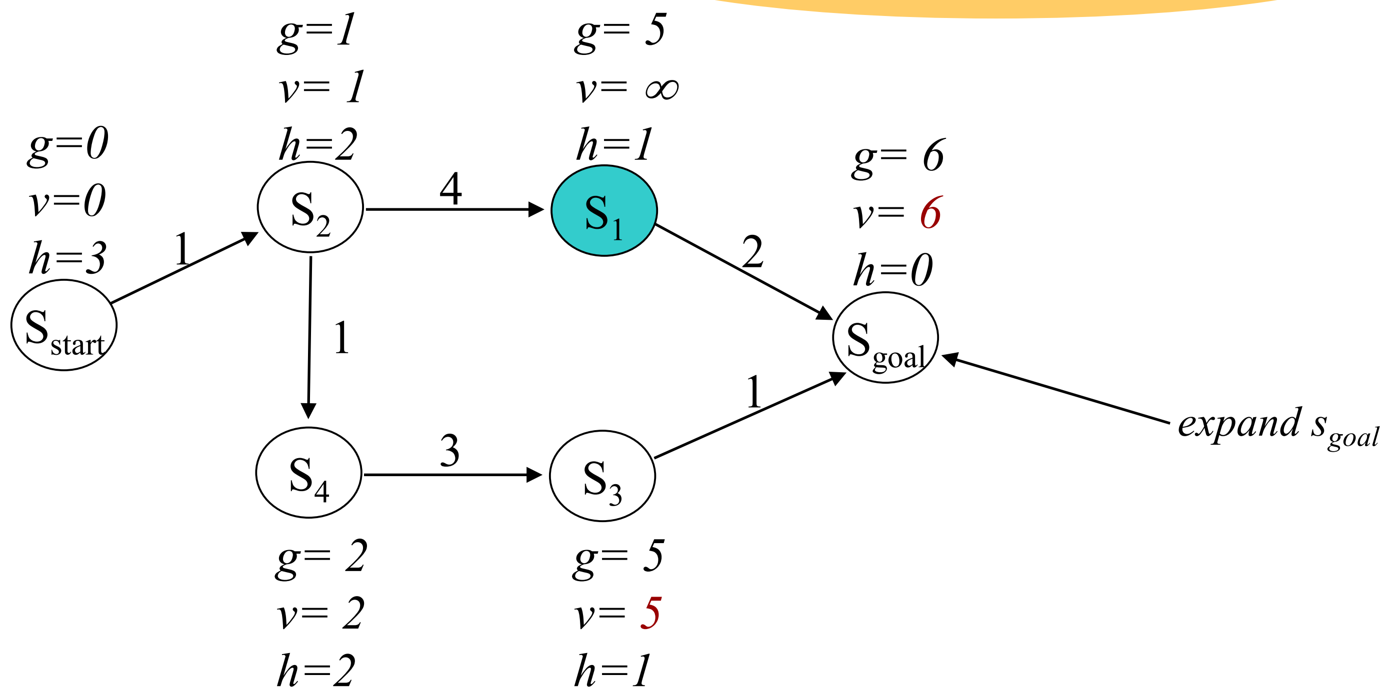


A* with Reuse of State Values

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

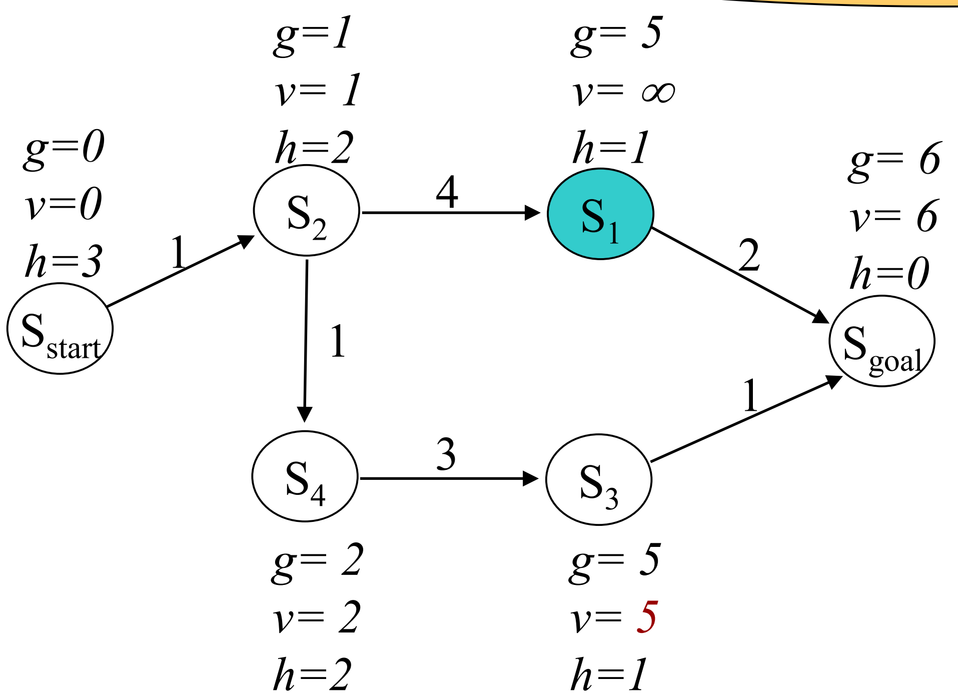


A* with Reuse of State Values

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

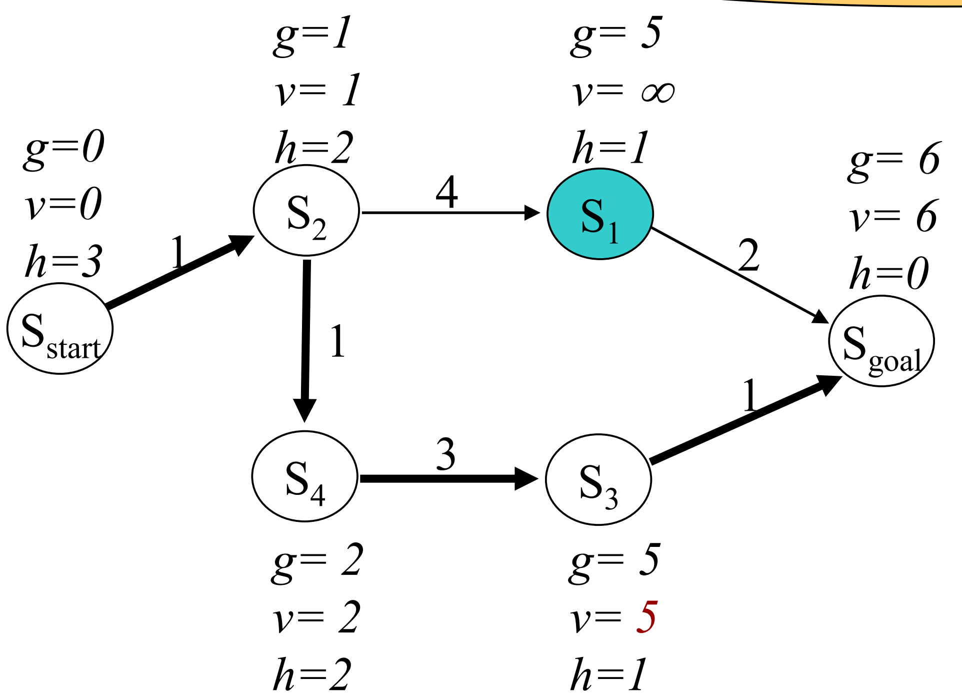


A* with Reuse of State Values

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

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?

Anytime Incremental Heuristic Search

- Anytime D* [Likhachev et al., 08]:
  - 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?

Other Uses of Incremental Heuristic Search

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

What You Should Know…

- The alternative formulation of A* that corresponds to a series of expansions of inconsistent states (states whose values are no longer consistent with their successors)

- How ARA* works

- What is an incremental search (D*/D* Lite) and when it is applicable and when it is not (i.e., its pros and cons)

- What is anytime incremental search (Anytime D*) and when it is applicable and when it is not (i.e., its pros and cons)