CSE-571

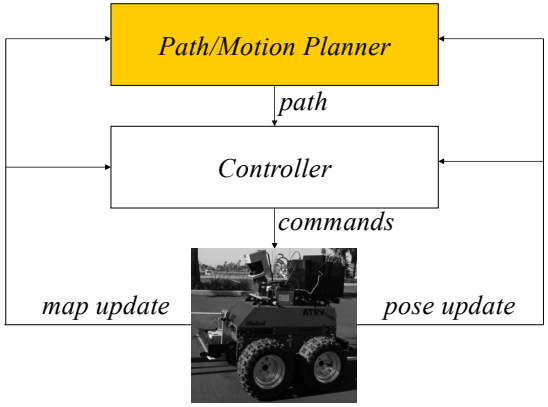
Deterministic Path Planning

Courtesy of Maxim Likhachev

Carnegie Mellon University

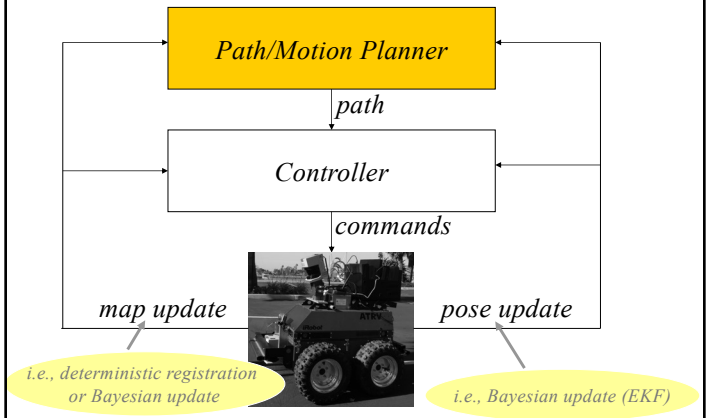
1

Motion/Path Planning



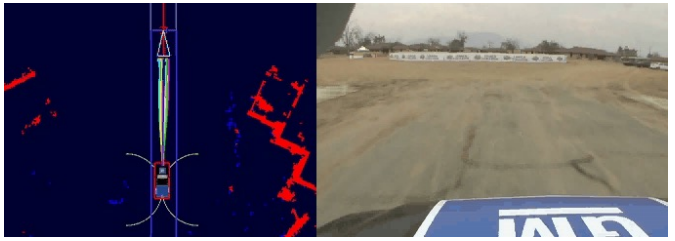
2

Motion/Path Planning



3

Example



*Urban Challenge Race, CMU team, planning with Anytime D**

4

Outline

- Deterministic planning
  - constructing a graph
  - search with A*
  - search with D*

5

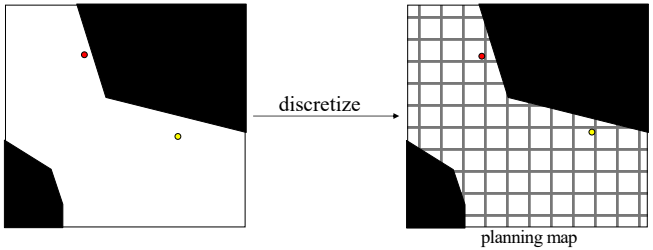
Outline

- Deterministic planning
  - constructing a graph
  - search with A*
  - search with D*

6

Planning via Cell Decomposition

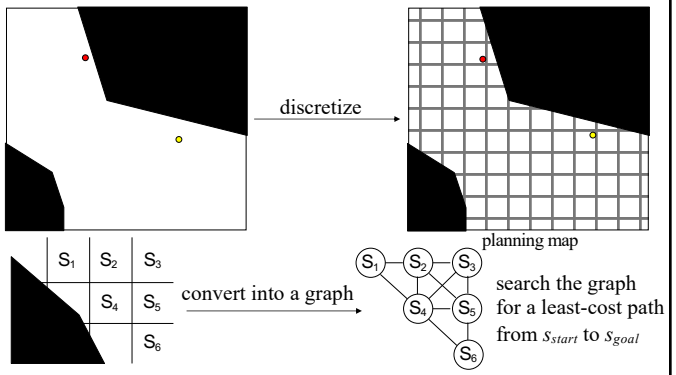
- Approximate Cell Decomposition:
  - overlay uniform grid over the C-space (discretize)



7

Planning via Cell Decomposition

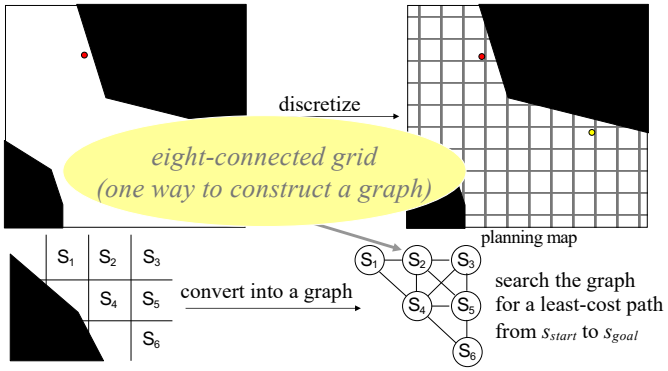
- Approximate Cell Decomposition:
  - construct a graph and search it for a least-cost path



8

Planning via Cell Decomposition

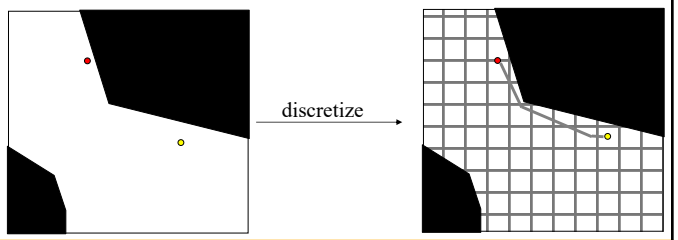
• Approximate Cell Decomposition:
  - construct a graph and search it for a least-cost path



9

Planning via Cell Decomposition

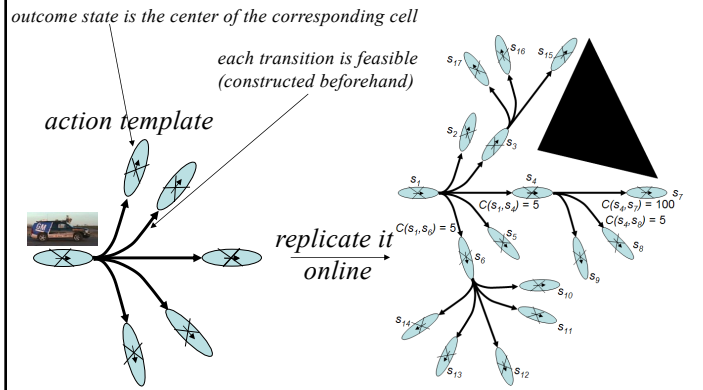
• Approximate Cell Decomposition:
  - construct a graph and search it for a least-cost path
    - VERY popular due to its simplicity and representation of arbitrary obstacles
    - Problem: transitions difficult to execute on non-holonomic robots



10

Planning via Cell Decomposition

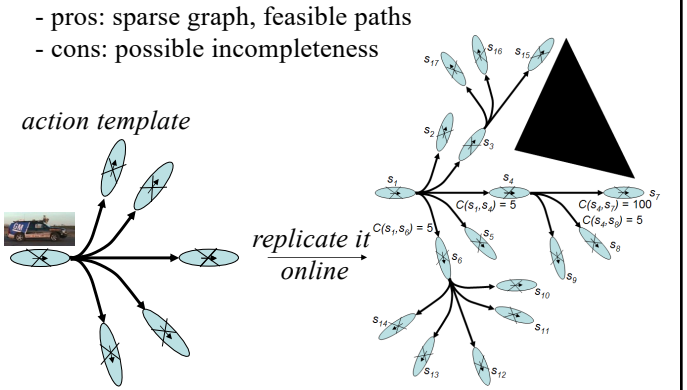
• Graph construction:
  - lattice graph



11

Planning via Cell Decomposition

• Graph construction:
  - lattice graph
  - pros: sparse graph, feasible paths
  - cons: possible incompleteness



12

Outline

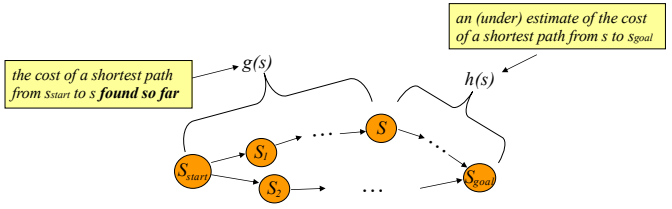
- Deterministic planning
  - constructing a graph
  - search with A*
  - search with D*
- Planning under uncertainty
  -Markov Decision Processes (MDP)
  -Partially Observable Decision Processes (POMDP)

13

A* Search

- Computes optimal g-values for relevant states

at any point of time:

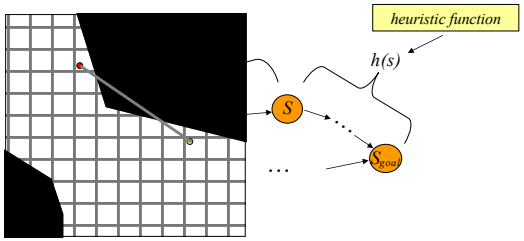


14

A* Search

- Computes optimal g-values for relevant states

at any point of time:



one popular heuristic function – Euclidean distance

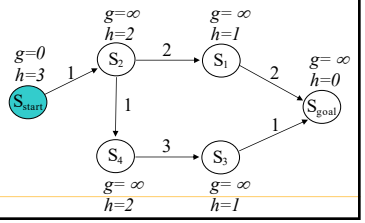
15

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 = {}
OPEN = {s_{start}}
next state to expand: s_{start}



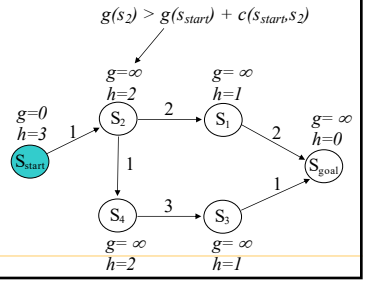
16

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 = {}
OPEN = {s_{start}}
next state to expand: s_{start}



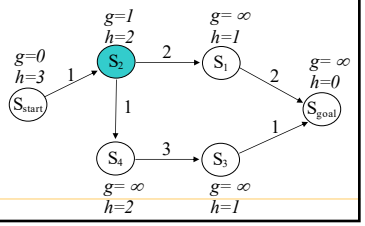
17

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}}
OPEN = {s_2}
next state to expand: s_2



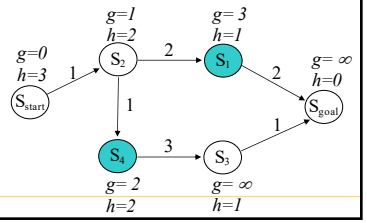
18

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



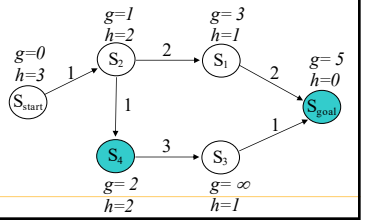
19

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



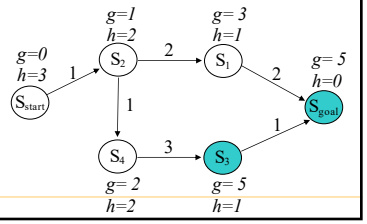
20

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}



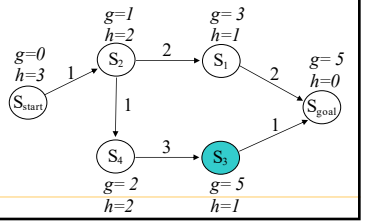
21

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



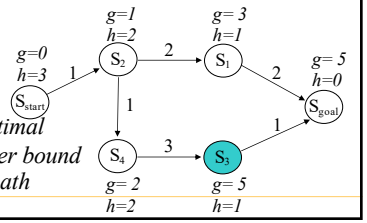
22

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



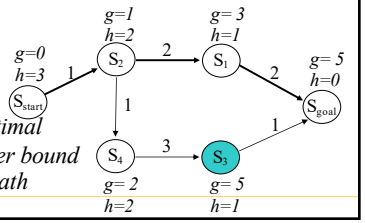
23

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*;

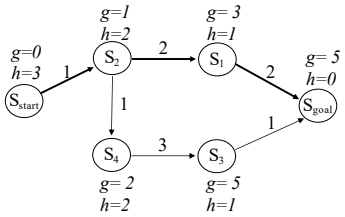
for every expanded state g(s) is optimal
for every other state g(s) is an upper bound
we can now compute a least-cost path



24

A* Search

- Is guaranteed to return an optimal path (in fact, for every expanded state) – optimal in terms of the solution
- Performs provably minimal number of state expansions required to guarantee optimality – optimal in terms of the computations

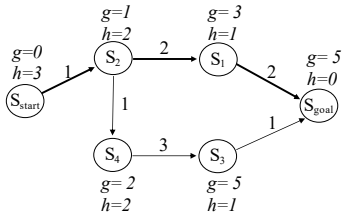


25

A* Search

- Is guaranteed to return an optimal path (in fact, for every expanded state) – optimal in terms of the solution
- Performs provably minimal number of state expansions required to guarantee optimality – optimal in terms of the computations

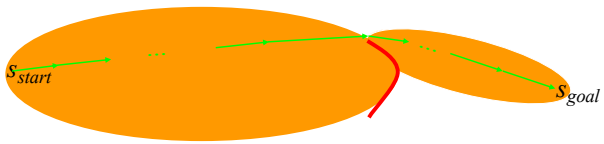
helps with robot deviating off its path
if we search with A backwards (from goal to start)*



26

Effect of the Heuristic Function

- A* Search: expands states in the order of $f = g+h$ values

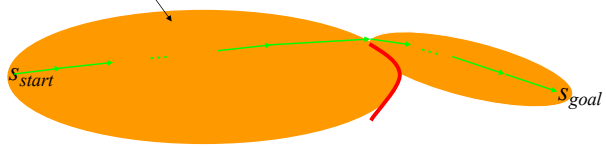


27

Effect of the Heuristic Function

- A* Search: expands states in the order of $f = g+h$ values

for large problems this results in A quickly running out of memory (memory: O(n))*

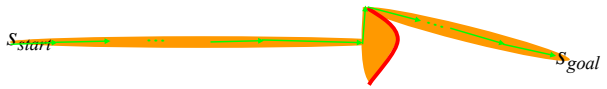


28

Effect of the Heuristic Function

- Weighted A* Search: expands states in the order of $f = g+\varepsilon h$ values, $\varepsilon > 1$ = bias towards states that are closer to goal

solution is always ε-suboptimal:
cost(solution) ≤ ε·cost(optimal solution)

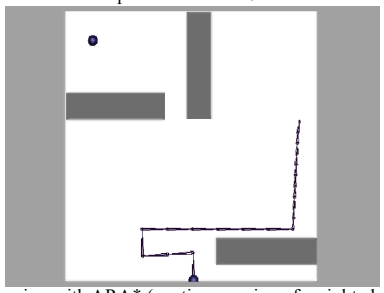


29

Effect of the Heuristic Function

- Weighted A* Search: expands states in the order of $f = g+\varepsilon h$ values, $\varepsilon > 1$ = bias towards states that are closer to goal

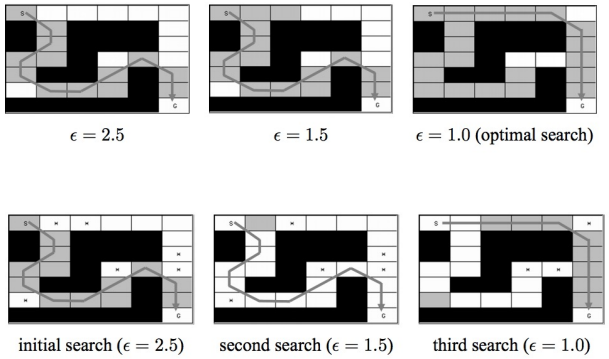
20DOF simulated robotic arm
state-space size: over 10^{26} states



planning with ARA* (anytime version of weighted A*)

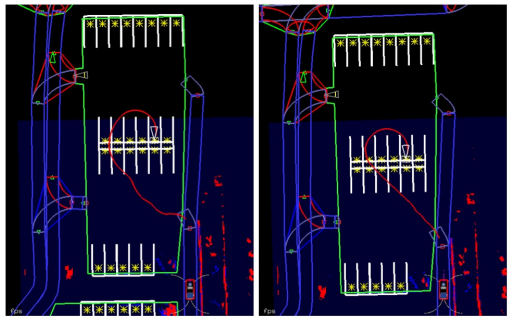
30

Adaptive Real-Time A*



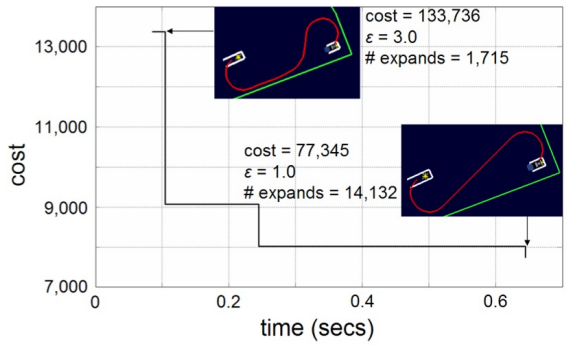
31

Anytime Aspects



32

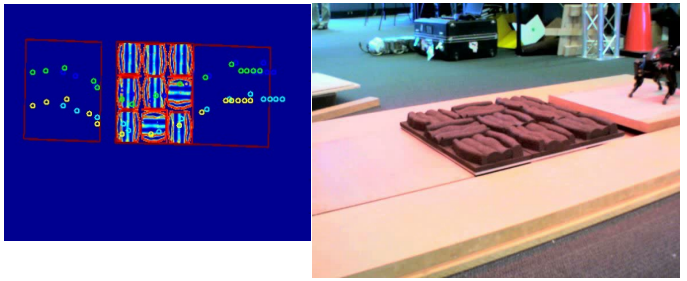
Anytime Aspects



33

Effect of the Heuristic Function

- planning in 8D (<x,y> for each foothold)
- heuristic is Euclidean distance from the center of the body to the goal location
- cost of edges based on kinematic stability of the robot and quality of footholds



planning with R* (randomized version of weighted A*)

joint work with Subhrajit Bhattacharya, Jon Bohren, Sachin Chitta, Daniel D. Lee, Aleksandr Kushleyev, Paul Vernaza

34

Outline

• Deterministic planning
  - constructing a graph
  - search with A*
  - search with D*

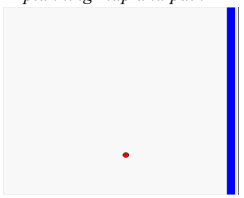
35

Incremental version of A* (D*/D* Lite)

- Robot needs to re-plan whenever
  - new information arrives (partially-known environments or/and dynamic environments)
  - robot deviates off its path

ATRV navigating initially-unknown environment

planning map and path

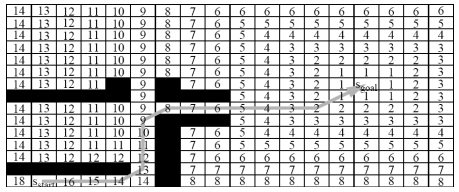


36

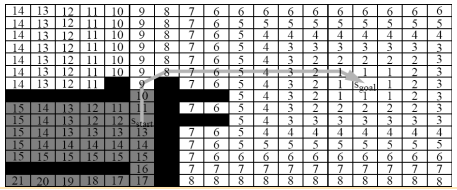
Motivation for Incremental Version of A*

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



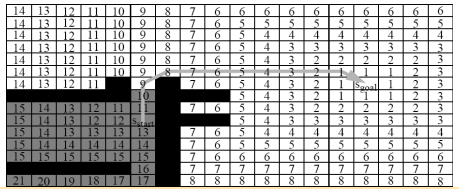
37

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



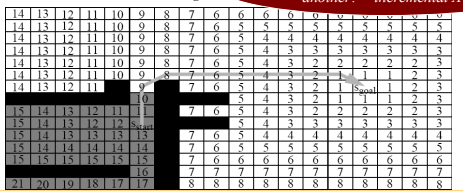
38

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}

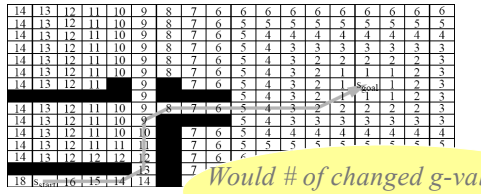


39

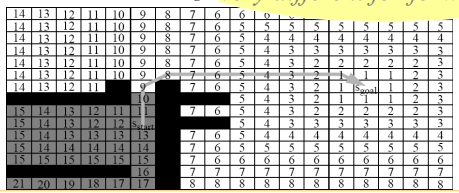
Motivation for Incremental Version of A*

Reuse state values from previous searches

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



cost of least-cost paths to s_{goal}

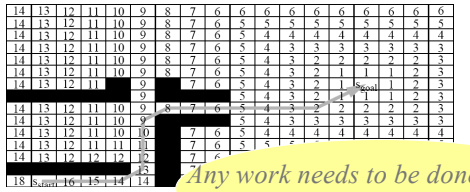


40

Motivation for Incremental Version of A*

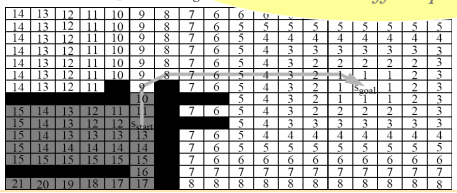
Reuse state values from previous searches

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



Any work needs to be done if robot deviates off its path?

cost of least-cost paths to s_{goal}

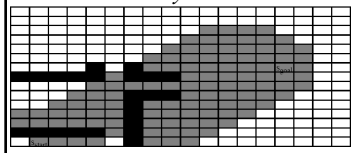


41

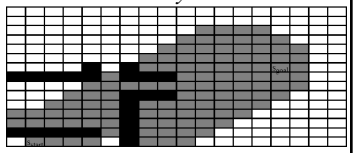
Incremental Version of A*

- Reuse state values from previous searches

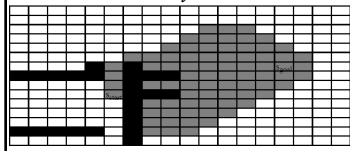
*initial search by backwards A**



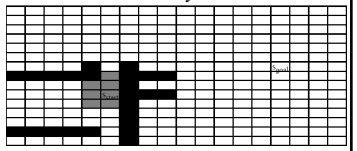
initial search by D Lite*



*second search by backwards A**



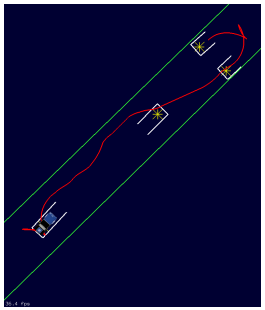
second search by D Lite*



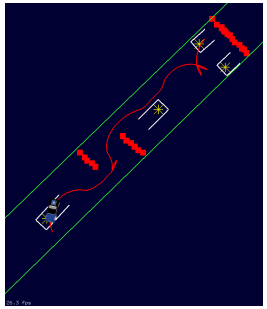
42

Searching the Graph

- Incremental behavior of Anytime D*:



initial path



a path after re-planning

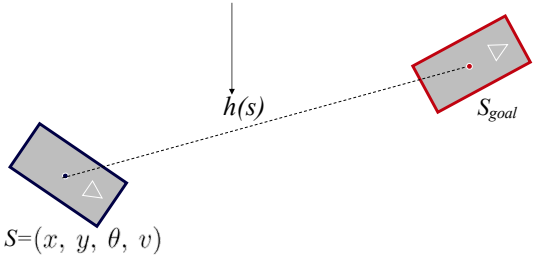
43

43

Searching the Graph

- Performance of Anytime D* depends strongly on heuristics *h(s)*: estimates of cost-to-goal

should be consistent and admissible (never overestimate cost-to-goal)



44

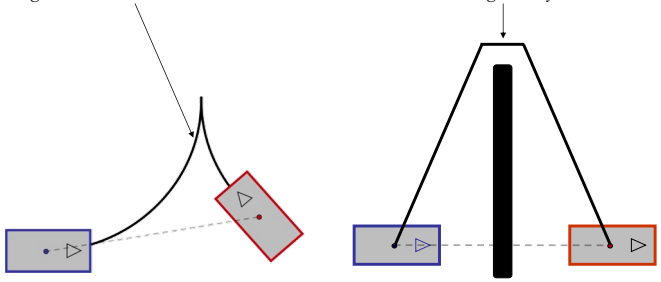
44

Searching the Graph

- In our planner: $h(s) = max(h_{mech}(s), h_{env}(s))$, where
  - $h_{mech}(s)$ – mechanism-constrained heuristic
  - $h_{env}(s)$ – environment-constrained heuristic

$h_{mech}(s)$ – *considers only dynamics constraints and ignores environment*

$h_{env}(s)$ – *considers only environment constraints and ignores dynamics*



45

45

Searching the Graph

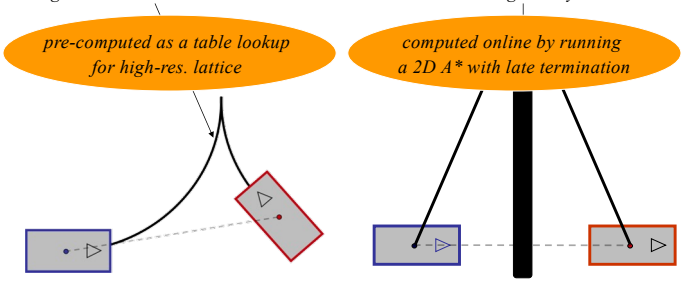
- In our planner: $h(s) = max(h_{mech}(s), h_{env}(s))$, where
  - $h_{mech}(s)$ – mechanism-constrained heuristic
  - $h_{env}(s)$ – environment-constrained heuristic

$h_{mech}(s)$ – *considers only dynamics constraints and ignores environment*

pre-computed as a table lookup for high-res. lattice

$h_{env}(s)$ – *considers only environment constraints and ignores dynamics*

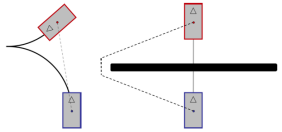
computed online by running a 2D A with late termination*

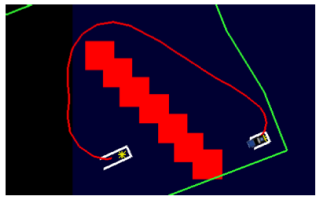


46

46

Heuristics

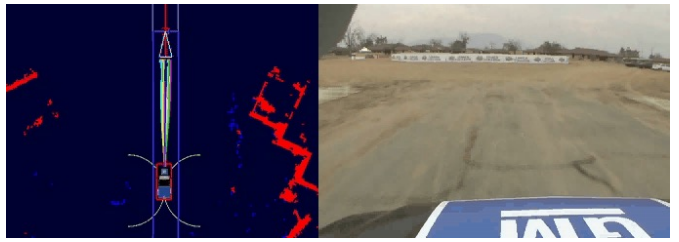




heuristic	states expanded	time (secs)
h	2,019	0.06
h_{2D}	26,108	1.30
h_{fsh}	124,794	3.49

47

Example, again



*Urban Challenge Race, CMU team, planning with Anytime D**

48

Summary

- Deterministic planning
  - constructing a graph
  - search with A*
  - search with D*

used a lot in real-time

think twice before trying to use it in real-time

- Planning under uncertainty
  - -Markov Decision Processes (MDP)
  - -Partially Observable Decision Processes (POMDP)

think three or four times before trying to use it in real-time

Many useful approximate solvers for MDP/POMDP exist!!

49

Manipulation Planning Examples

50

Maxim Likhachev on A*/D* vs PRM/RRT

- 1. Sampling-based methods are typically much easier to get working. One of the great thing about RRT is that it doesn't require careful discretization of the action space and instead takes advantage of an extend operator (i.e., local controller or an interpolation function) which naturally exists in most robotics systems
- 2. For planning in a continuous space, when comparing a quick implementation of RRT and a quick implementation of Anytime version of A*, RRT is typically much faster due to sparse exploration of a space.
- 3. A* and its variants are typically harder to implement because they require a) careful design of discretization of the state-space and action-space (to make sure edges land where they are supposed to land); b) careful design of the heuristic function to guide the search well.

51

Maxim Likhachev on A*/D* vs PRM/RRT

- 4. A* and its variants (including anytime variants) typically generate better quality solutions and very consistent solutions (similar solutions for similar queries) which is beneficial in many domains.
- 5. A* and its variants can often be made nearly as fast as RRT and sometimes even faster if one analyzes the robotic system well to derive a powerful heuristic function. Many robotic systems have natural low-dimensional manifolds (e.g., a 3D workspace for example) that can be used to derive such heuristic functions.
- 6. A* and its variants can be applied to both discrete and continuous (as well as hybrid) systems, whereas sampling-based systems tend to be more suitable for continuous systems since they rely on the idea of sparse exploration. (Within the same point, it should be noted that A* and its variants apply to PRMs and its variants. PRM is just a particular graph representation of the environment.)

52

Maxim Likhachev on A*/D* vs PRM/RRT

- Anyway, in summary, I think for continuous planning problems, A* and its variants require substantially more development efforts (careful analysis of the system to derive proper graph representation and a good heuristic function) but can result in a better performance (similar speed but better quality solutions and more consistent behavior).

53