CSE-571

Deterministic Path Planning in Robotics

Courtesy of Maxim Likhachev

Carnegie Mellon University

Motion/Path Planning

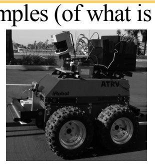
- Task:
  find a feasible (and cost-minimal) path/motion from the current configuration of the robot to its goal configuration (or one of its goal configurations)

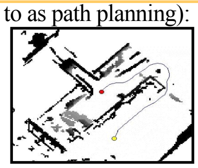
- Two types of constraints:
  environmental constraints (e.g., obstacles)
  dynamics/kinematics constraints of the robot

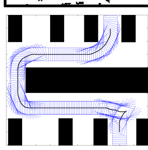
- Generated motion/path should (objective):
  be any feasible path
  minimize cost such as distance, time, energy, risk, …

Motion/Path Planning

Examples (of what is usually referred to as path planning):

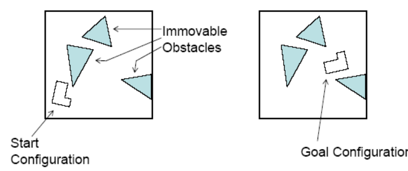






Motion/Path Planning

Examples (of what is usually referred to as motion planning):

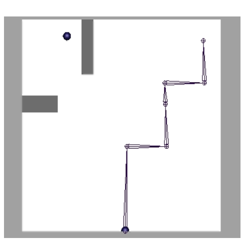


Piano Movers' problem

the example above is borrowed from www.cs.cmu.edu/~awm/tutorials

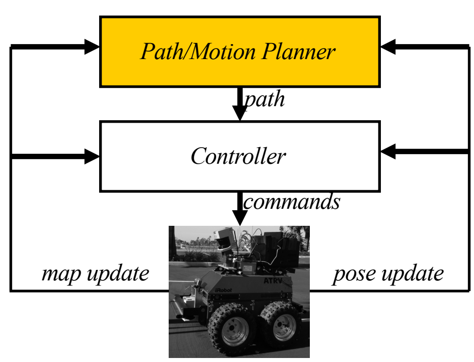
Motion/Path Planning

Examples (of what is usually referred to as motion planning):

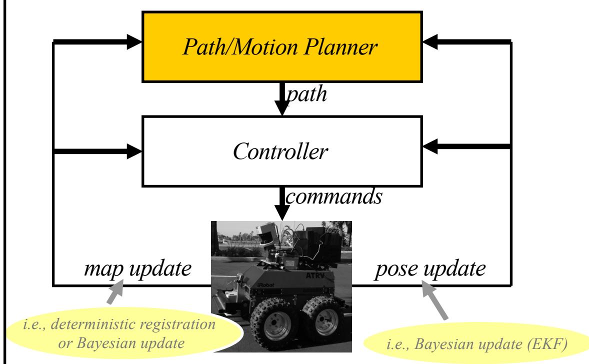


Planned motion for a 6DOF robot arm

Motion/Path Planning



Motion/Path Planning



Uncertainty and Planning

- Uncertainty can be in:
  - prior environment (i.e., door is open or closed)
  - execution (i.e., robot may slip)
  - sensing environment (i.e., seems like an obstacle but not sure)
  - pose
- Planning approaches:
  - deterministic planning:
    - assume some (i.e., most likely) environment, execution, pose
    - plan a single least-cost trajectory under this assumption
    - re-plan as new information arrives
  - planning under uncertainty:
    - associate probabilities with some elements or everything
    - plan a policy that dictates what to do for each outcome of sensing/action and minimizes expected cost-to-goal
    - re-plan if unaccounted events happen

Uncertainty and Planning

- Uncertainty can be in:
  - prior environment (i.e., door is open or closed)
  - execution (i.e., robot may slip)
  - sensing environment (i.e., seems like an obstacle but not sure)
  - pose
- Planning approaches:
  - deterministic planning:
    - assume some (i.e., most likely) environmen
    - plan a single least-cost trajectory under thi
    - re-plan as new information arrives
  - planning under uncertainty:
    - associate probabilities with some elements or everything
    - plan a policy that dictates what to do for each outcome of sensing/action and minimizes expected cost-to-goal
    - re-plan if unaccounted events happen

re-plan every time sensory data arrives or robot deviates off its path

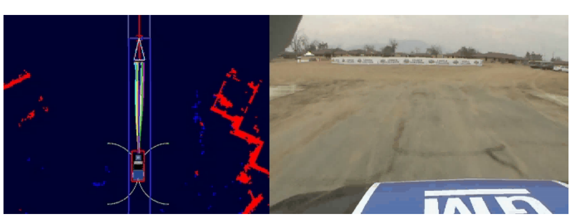
re-planning needs to be FAST

Uncertainty and Planning

- Uncertainty can be in:
  - prior environment (i.e., door is open or closed)
  - execution (i.e., robot may slip)
  - sensing environment (i.e., seems like an obstacle but not sure)
  - pose
- Planning approaches:
  - deterministic planning:
    - assume some (i.e., most likely) environment, execution, pose
    - plan a single least-cost trajectory under this assumption
    - re-plan as new information arrives
  - planning under uncertainty:
    - associate probabilities with some elements or everything
    - plan a policy that dictates what to do for each outcome of sensing/action and minimizes expected cost-to-goal
    - re-plan if unaccounted events happen

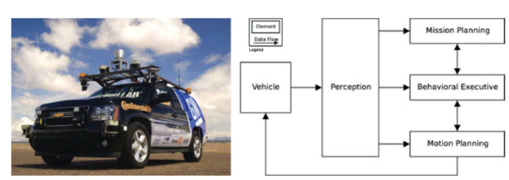
computationally MUCH harder

Example



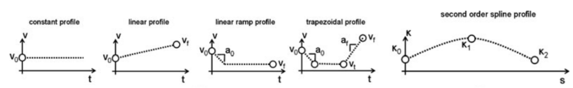
*Urban Challenge Race, CMU team, planning with Anytime D**

Control Architecture of Boss

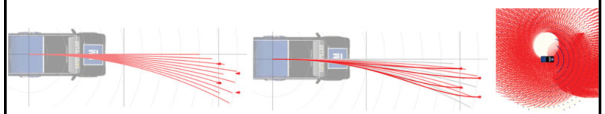


Representing Motion Commands

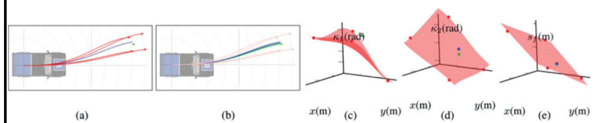
- State is position, curvature, and velocity
  $< x, y, \theta, \kappa, v >$
- Velocity profiles determine forward speed
  $< v_0, v_t, v_f, a_0, a_f, t_0, t_f >$
- Spline represents shape and overall time
  $< \kappa_0, \kappa_1, \kappa_2, s_f >$



Trajectory Pre-Computation and Optimization

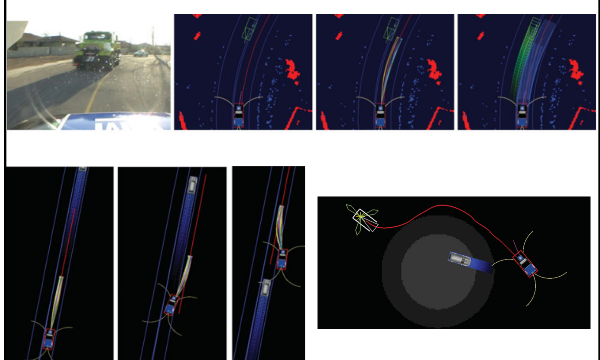


Pre-compute parameters for set of end points

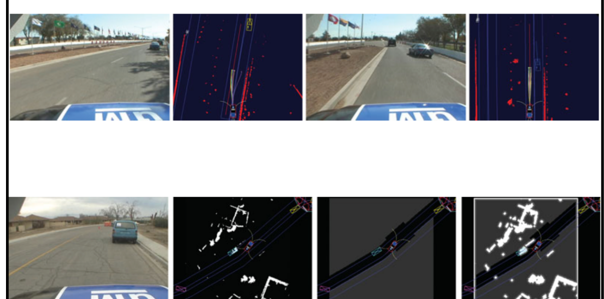


Optimize (fine-tune) parameters initialized via interpolation

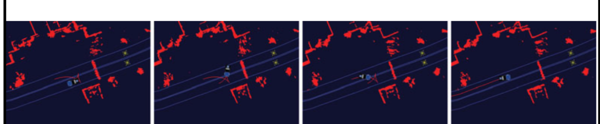
Predicting and Avoiding Other Vehicles



Passing and Cost



U-Turns



Outline

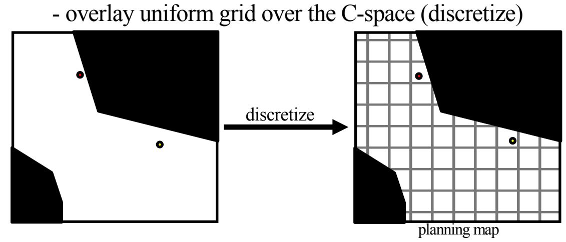
- Deterministic planning
  - constructing a graph
  - search with A*
  - search with D*

Outline

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

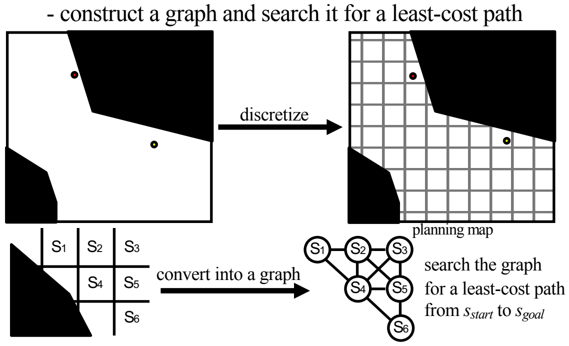
Planning via Cell Decomposition

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



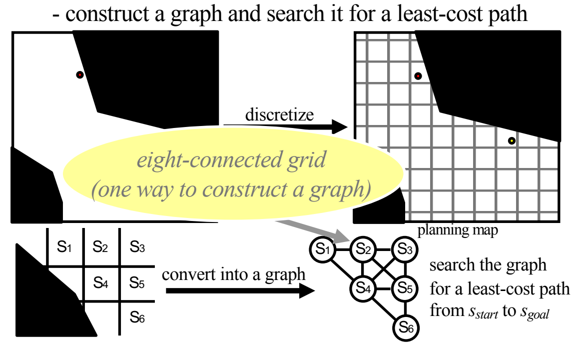
Planning via Cell Decomposition

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



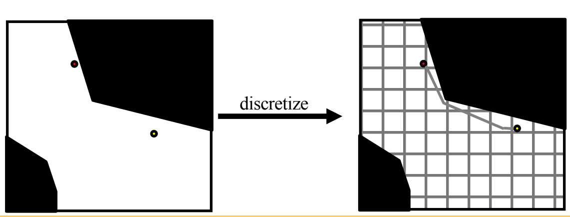
Planning via Cell Decomposition

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



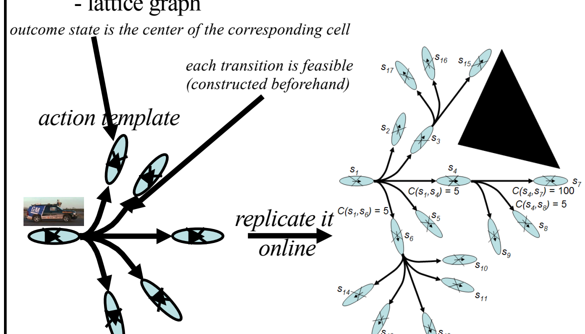
Planning via Cell Decomposition

- 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



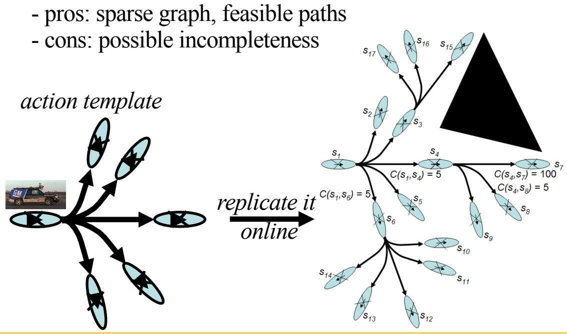
Planning via Cell Decomposition

- Graph construction:
  - lattice graph



Planning via Cell Decomposition

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



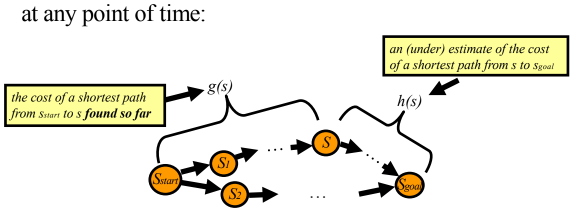
Outline

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

A* Search

- Computes optimal g-values for relevant states

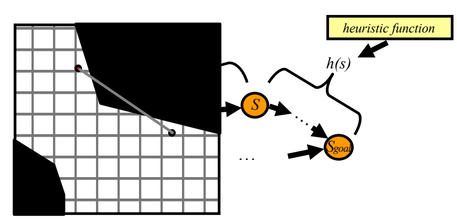
at any point of time:



A* Search

- Computes optimal g-values for relevant states

at any point of time:



one popular heuristic function – Euclidean distance

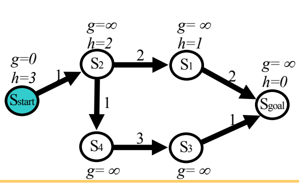
A* Search

- 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}



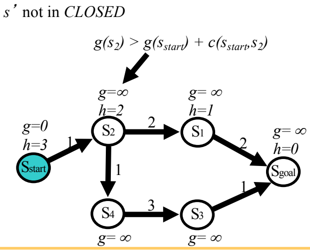
A* Search

- 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}



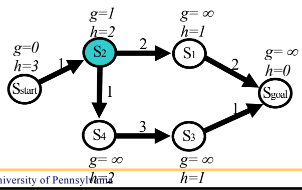
A* Search

- 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



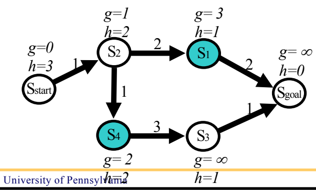
A* Search

- 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



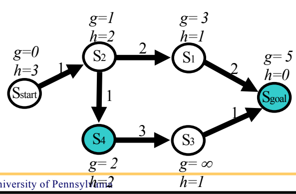
A* Search

- 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



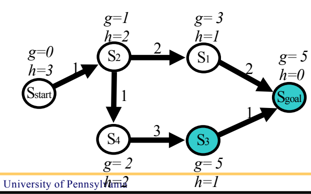
A* Search

- 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}



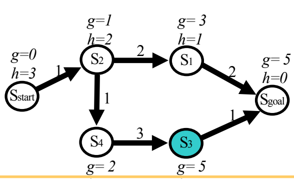
A* Search

- 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



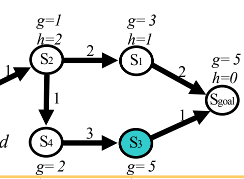
A* Search

- 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

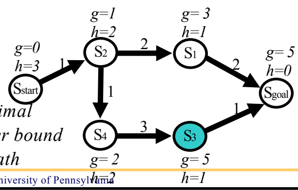


A* Search

- Computes optimal g-values for relevant states

ComputePath function

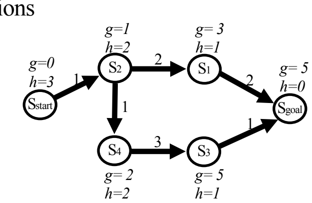
```
while(s_goal is not expanded)
  remove s with the smallest [f(s) = g(s)+h(s)] from OPEN;
  insert s into CLOSED;
  for every successor s' of s such that s' not in CLOSED
    if g(s') > g(s) + c(s,s')
      g(s') = g(s) + c(s,s');
      insert s' into OPEN;
```



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

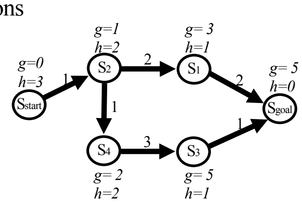
- Is guaranteed to return an optimal path (in fact, for every expanded state) – optimal in terms of the solution
- Performs provably minimal number of state expansions required to guarantee optimality – optimal in terms of the computations



A* Search

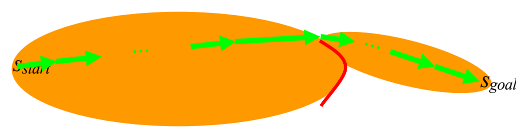
- Is guaranteed to return an optimal path (in fact, for every expanded state) – optimal in terms of the solution
- Performs provably minimal number of state expansions required to guarantee optimality – optimal in terms of the computations

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



Effect of the Heuristic Function

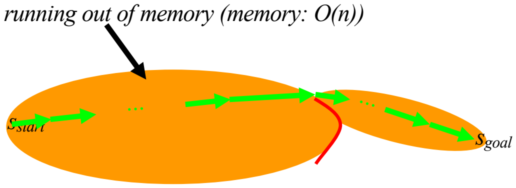
- A* Search: expands states in the order of $f = g+h$ values



Effect of the Heuristic Function

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

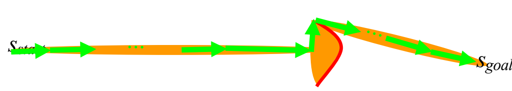
for large problems this results in A quickly running out of memory (memory: O(n))*



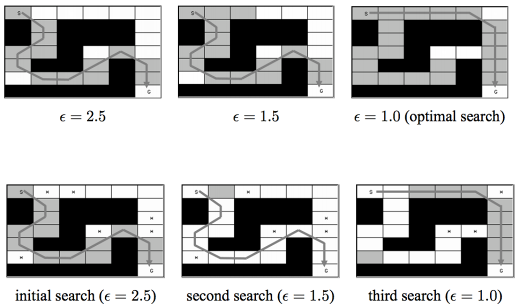
Effect of the Heuristic Function

- 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)



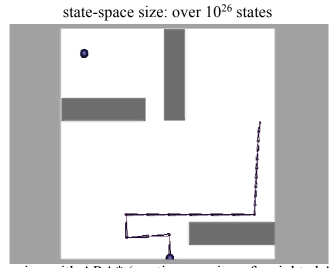
Adaptive Real-Time A*



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

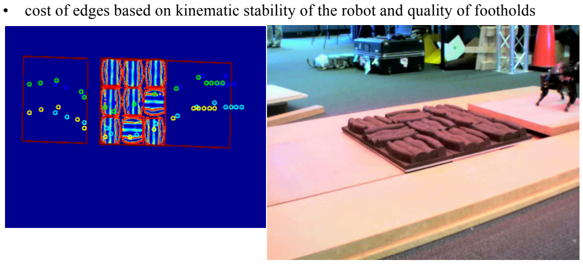
20DOF simulated robotic arm
state-space size: over 10^{26} states



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

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

Outline

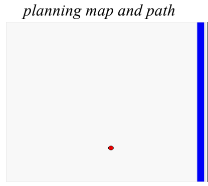
- Deterministic planning
  - constructing a graph
  - search with A*
  - search with D*

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

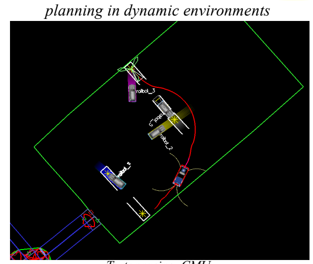


Incremental version of A* (D*/D* Lite)

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

***incremental planning (re-planning)**: reuse of previous planning efforts*

planning in dynamic environments

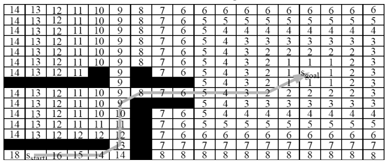


Tartanracing, CMU

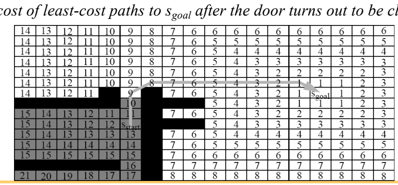
Motivation for Incremental Version of A*

- 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

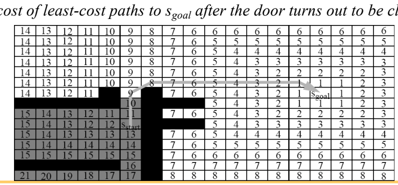


Motivation for Incremental Version of A*

- Reuse state values from previous searches

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

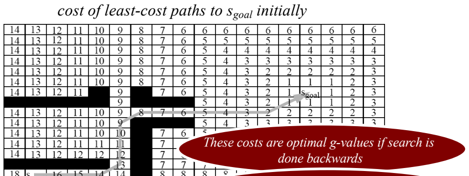
cost of least-cost paths to s_{goal} after the door turns out to be closed



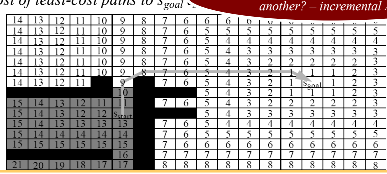
Motivation for Incremental Version of A*

- Reuse state values from previous searches

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



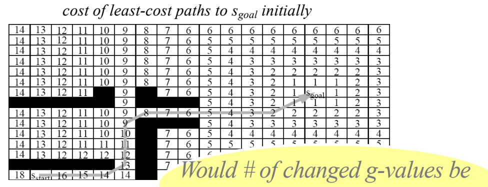
cost of least-cost paths to s_{goal}



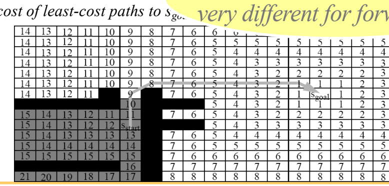
Motivation for Incremental Version of A*

- Reuse state values from previous searches

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



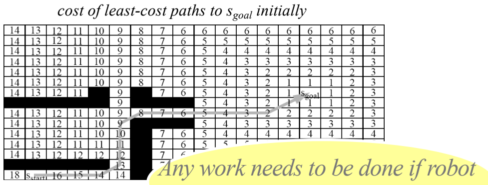
cost of least-cost paths to s_{goal}



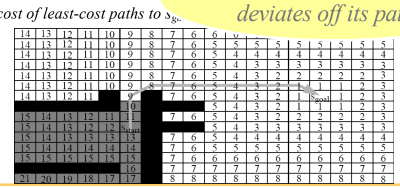
Motivation for Incremental Version of A*

- Reuse state values from previous searches

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



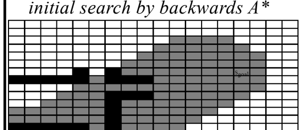
cost of least-cost paths to s_{goal}



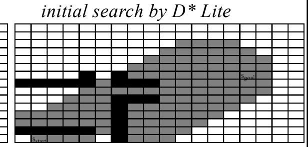
Incremental Version of A*

- Reuse state values from previous searches

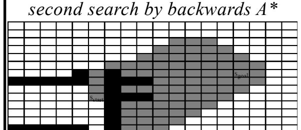
*initial search by backwards A**



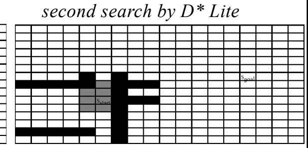
initial search by D Lite*



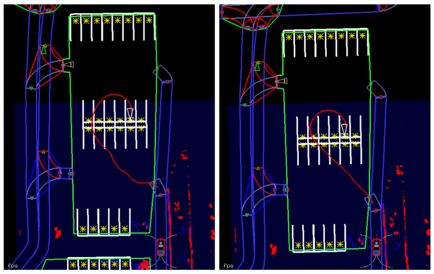
*second search by backwards A**



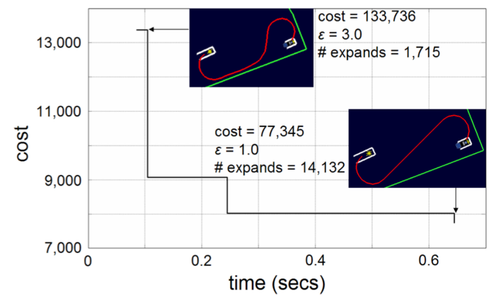
second search by D Lite*



Anytime Aspects

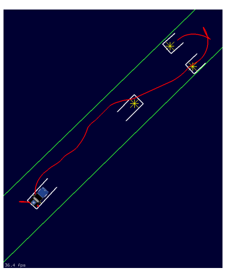


Anytime Aspects

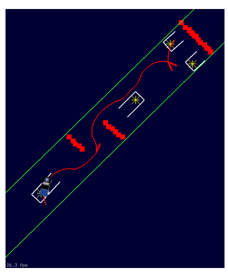


Searching the Graph

- Incremental behavior of Anytime D*:



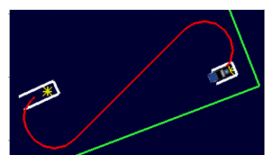
initial path



a path after re-planning

Building the Graph

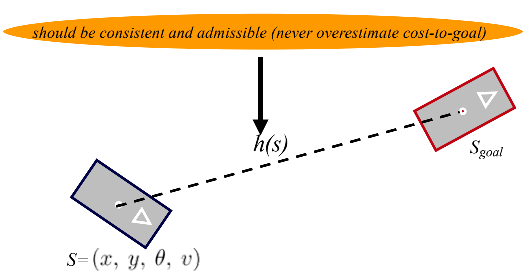
- Benefit of the multi-resolution lattice used for Urban Challenge:



Lattice	States Expanded	Planning Time (s)
High-resolution	2,933	0.19
Multi-resolution	1,228	0.06

Searching the Graph

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

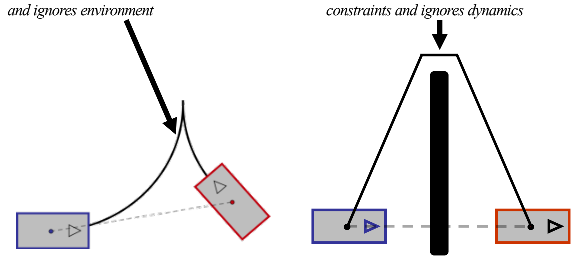


Searching the Graph

- 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

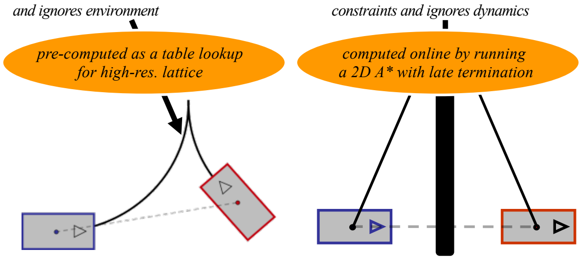


Searching the Graph

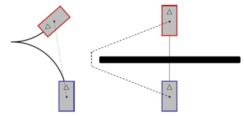
- In our planner: $h(s) = max(h_{mech}(s), h_{env}(s))$, where
  - $h_{mech}(s)$ – mechanism-constrained heuristic
  - $h_{env}(s)$ – environment-constrained heuristic

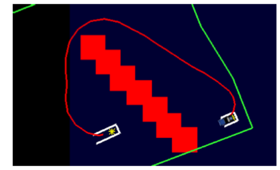
$h_{mech}(s)$ – considers only dynamics constraints and ignores environment

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



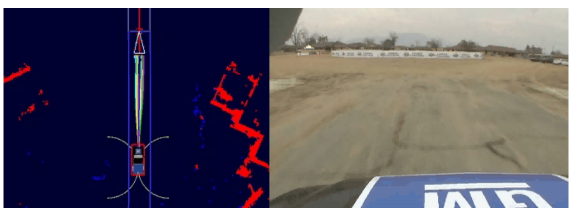
Heuristics





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

Example, again



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

Google Now

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!!