CS 4649/7649  
Robot Intelligence: Planning

Roadmap Approaches

Sungmoon Joo  
School of Interactive Computing  
College of Computing  
Georgia Institute of Technology

Course Info.

- HW#1 due Oct 6th
- Wiki - Add your group info.
- Need a repo.?
- Late policy – No late HWs
**Navigation Planning**

Assuming full knowledge, how does the robot ‘plan’ its path from S to G?

**Visibility Graph**

- Assuming polygonal obstacles, it looks like the shortest path is a sequence of straight lines joining the vertices of the obstacles
• Assuming polygonal obstacles, it looks like the shortest path is a sequence of straight lines joining the vertices of the obstacles.

• Visibility graph $G = \text{set of unblocked lines between vertices of the obstacles } + \text{ initial & goal configuration}$

  $$= (V, E), V = \{\text{obstacle vertices}\} \cup \{\text{start, goal}\},$$

  $$E = \{\text{edges } (v_i, v_j) \text{ disjoint from obstacle interiors}\}$$

**Visibility Graphs**

• Early Motion Planning Algorithm

• Nodes share an edge if they are within "line of sight"

• All points in free space are within sight of at least one node

* "An algorithm for planning collision-free paths among polyhedral obstacles" 1979 T. Lozano-Perez & M. A. Wesley

* "A mobile automaton: An application of artificial intelligence techniques" 1969 N.J Nilson
Path Planning for Robots with Geometric Shapes

Visibility Graph Analysis

- Visibility Graphs are complete? Yes (Assuming Polygonal Obstacles)
- Visibility Graphs are optimal? Yes Metric: Distance Traveled
Roadmap Approach to Navigation Planning

- Assumption: Static environment
- General Idea:
  - Avoid searching the entire space
  - Pre-compute a (hopefully small) graph (i.e. the roadmap) s.t. staying on the roads is guaranteed to avoid the ‘obstacles’ (& to take us to the goal)
  - Search a path between \(q_{\text{init}}\) and \(q_{\text{goal}}\) on the roadmap

Visibility Graph is an example of the roadmap approach

Assumptions/Limitations? Polygonal objects, Zero distance to obstacles

Do we really care about strict optimality with distance metric?
What about safety?
Voronoi Diagrams

- Another type of ‘roadmap’
- Lets color this space
- How would you distribute the continuous space into 4 colors?

Voronoi diagram is a way of dividing space into a number of regions

Voronoi Diagrams

- Space Coloring
  - Associate each point with the area closest to it
  - Boundaries have maximum distance to points

Gregory Voronoi 1868-1908 ... student of Andrey Markov (sound familiar?)
- Voronoi Diagram = The set of line segments separating the regions corresponding to different colors
Voronoi Diagrams

- Voronoi Diagram = The set of line segments separating the regions corresponding to different colors

Complexity (in 2D plane) – $O(n \log n)$ in time, $O(n)$ in space

Gregory Voronoi 1868-1908 ... student of Andrey Markov (sound familiar?)

- Voronoi Diagram = The set of line segments separating the regions corresponding to different colors
Generalized Voronoi Diagram (GVD)

Extending to polygonal objects:

- **Features** of an obstacle are vertices and edges
- A roadmap edge must be between some pair of features

---

Generalized Voronoi Diagram (GVD)

The points on the edges are the furthest from the obstacles & workspace boundary!!
Voronoi Diagram → Navigation Planning

- Idea: Construct a path between $q_{init}$ and $q_{goal}$ by following edges on the Voronoi diagram

Voronoi diagram = Roadmap

Step1. Find the point $q^*_init$ of the Voronoi diagram closest to $q_{init}$

Step2. Find the point $q^*_goal$ of the Voronoi diagram closest to $q_{goal}$

Step3. Compute shortest path from $q^*_init$ to $q^*_goal$ on the Voronoi diagram

Practical Bug Alternative using GVD

Online Diagram Generation:
- Locate two nearest obstacles and move to midpoint
- Continue along GVG edge until
  - Detect new GVG edges
  - Select one to explore and continue
Practical Bug Alternative using GVD

Online Diagram Generation:
- Locate two nearest obstacles and move to midpoint
- Continue along GVG edge until
  - Detect new GVG edges
  - Select one to explore and continue
- Detect boundary points and cycles

Bushfire Algorithm to Generate GVD

Bushfire GVG Generation
- "Wavefront"
- Start with an empty grid with obstacles = 1
Bushfire Algorithm to Generate GVD

Bushfire GVG Generation
• "Wavefront"
• Start with an empty grid with obstacles = 1
• Expand all cells (i) to (i+1)
• If a cell is expanded twice, label as a GVG edge and do not expand further.
Bushfire Algorithm to Generate GVD

Bushfire GVG Generation

- "Wavefront"
- Start with an empty grid with obstacles = 1
- Expand all cells (i) to (i+1)
- If a cell is expanded twice, label as a GVG edge.

Voronoi Diagrams: Analysis

Completeness:
Yes for Polygonal obstacles

Optimality: ?
Voronoi Diagrams: Summary

- Popular Motion Planning Algorithm
- Edges maximally separate domain features
- Nodes are critical points where edges intersect
- Many practical variants in existence

Voronoi Diagrams: Summary

- Difficult in higher dimensions or nonpolygonal worlds
- Use of Voronoi is not necessarily the best heuristic ("stay away from obstacles")
  - Can be much too conservative
  - Can be unstable:
    Small changes in $C_{\text{obstacle}} \rightarrow$ Large changes in GVD
Summary

Roadmap Approach
• Static environment
• Avoid searching the entire space
  - Pre-compute a graph (i.e. roadmap) → Search space reduction

Roadmap approach for Navigation Planning
  - Visibility Graph → Short path
  - Voronoi Diagram → Safe/Conservative path

Other Options

• Exact Cell Decomposition

• Approximate Decomposition

• Potential Fields (Not really grids, but relevant)

• Practical matters: Re-planning
Exact Cell Decomposition: Convex Polygons

- Collection of non-overlapping cells: \( \text{Union(Cells)} = \text{Free Space} \)
- Finite set of convex polygons that cover the space

"The graph of midpoints of edges between adjacent cells defines a roadmap"
Exact Cell Decomposition: Trapezoidal

- Collection of non-overlapping cells: $\text{Union} (\text{Cells}) = \text{Free Space}$

Extend a bi-directional vertical line from each vertex until collision
- Gives you trapezoids
- How do we search?

---

Exact Cell Decomposition: Trapezoidal

- Collection of non-overlapping cells: $\text{Union} (\text{Cells}) = \text{Free Space}$

Extend a bi-directional vertical line from each vertex until collision
- This is a convex polygonal decomposition
- Again a graph search
**Exact Cell Decomposition: Trapezoidal**

- How do we compute this efficiently?
- Plane Sweep

Critical events: Create new cell, Split cell, Merge cells

S. Joo (sungmoon.joo@cc.gatech.edu) 9/30/2014 33

---

**Exact Cell Decomposition: Trapezoidal**

- How do we compute this efficiently?
- Plane Sweep

S. Joo (sungmoon.joo@cc.gatech.edu) 9/30/2014 34
Exact Cell Decomposition: Trapezoidal

- How do we compute this efficiently?
- Plane Sweep

S. Joo (sungmoon.joo@cc.gatech.edu) 9/30/2014 35
Exact Cell Decomposition: Trapezoidal

- How do we compute this efficiently?
- Plane Sweep

Complexity (in 2D plane)
- Time: $O(n \log n)$
- Space: $O(n)$
**Exact Cell Decomposition: Analysis**

- Complete? Yes
- Optimal? ?
- Advantage? Efficiency!*

*Expensive and difficult to implement in higher dimensions

**Approximate Cell Decomposition**

- Use grid
- Is it complete?
  - Yes, up to grid size
  - Resolution Complete
- Is it optimal?
  - Metric = # grid cells traversed
  - Search Method = A*
  - Yes IF heuristic is admissible
Better Approximate Cell Decomposition

- $2^m$ Trees
- $m =$ number of cell subdivisions
- Only Divide Mixed Cells!

m = 2
$2^2 = 4$
QuadTree

Better Approximate Cell Decomposition

- $2^m$ Trees
- $m =$ number of cell subdivisions
- Only Divide Mixed Cells!

m = 3
$2^3 = 8$
OctTree
Remember BFS: Policy?

Not a Policy!
BFS / Dijkstra CAN make a policy!

How?
Dijkstra & A*

A* - Cost = Optimal Cost-to-Come + Heuristic of Cost-to-Go
Dijkstra - Cost = Optimal Cost-to-Come

- If a heuristic of 'Cost-to-Go' becomes closer to the true optimal cost-to-go, fewer vertices tend to be explored in comparison with Dijkstra's algorithm.
- Dijkstra = A* with Zero-Cost-to-Go (degenerate case of A*)
### Policy

State $\rightarrow$ Action

We know what to do at each cell $\rightarrow$ Policy!

### Why is this useful - Replanning
**D* / D* Lite Concept**

- Maintains cost-to-go values
- Propagate backward from the goal → Backward Dijkstra
- Useful for navigation in an unknown environment