Advanced Games Programming (AI) Part 2: Advanced Path Planning Topics

Michael Buro

December 23, 2024

[Under Construction]

Change Log

- [Nov 07] Added book triangulation book reference [12]
- [Oct 06] Created

Outline

1. Hierarchical Pathfinding A*
2. Path Refinement A*
3. Triangulations
4. Point Localization
5. Dynamic Constraint Delaunay Triangulations
6. Triangulation-Based Path Planning
7. Triangulation Reduction
8. Experimental Results

[AI-Lec 7 L11] Advanced Path Planning Topics

- Path Planning in Abstractions
- Hierarchical Path Planning A* (HPA*)
- Path-Refinement A* (PRA*)
- Triangulation A* (TA*)
- Triangulation Reduction Path Planning (TRA*)

Goal:

Use abstractions to speed up A* search while still producing accurate solutions on large maps

Recurring Idea:

Build smaller abstract representation of search graph which maintains important propertiers of the original topology

Hierarchical Pathfinding A* (HPA*, [2])

A. Pre-processing:

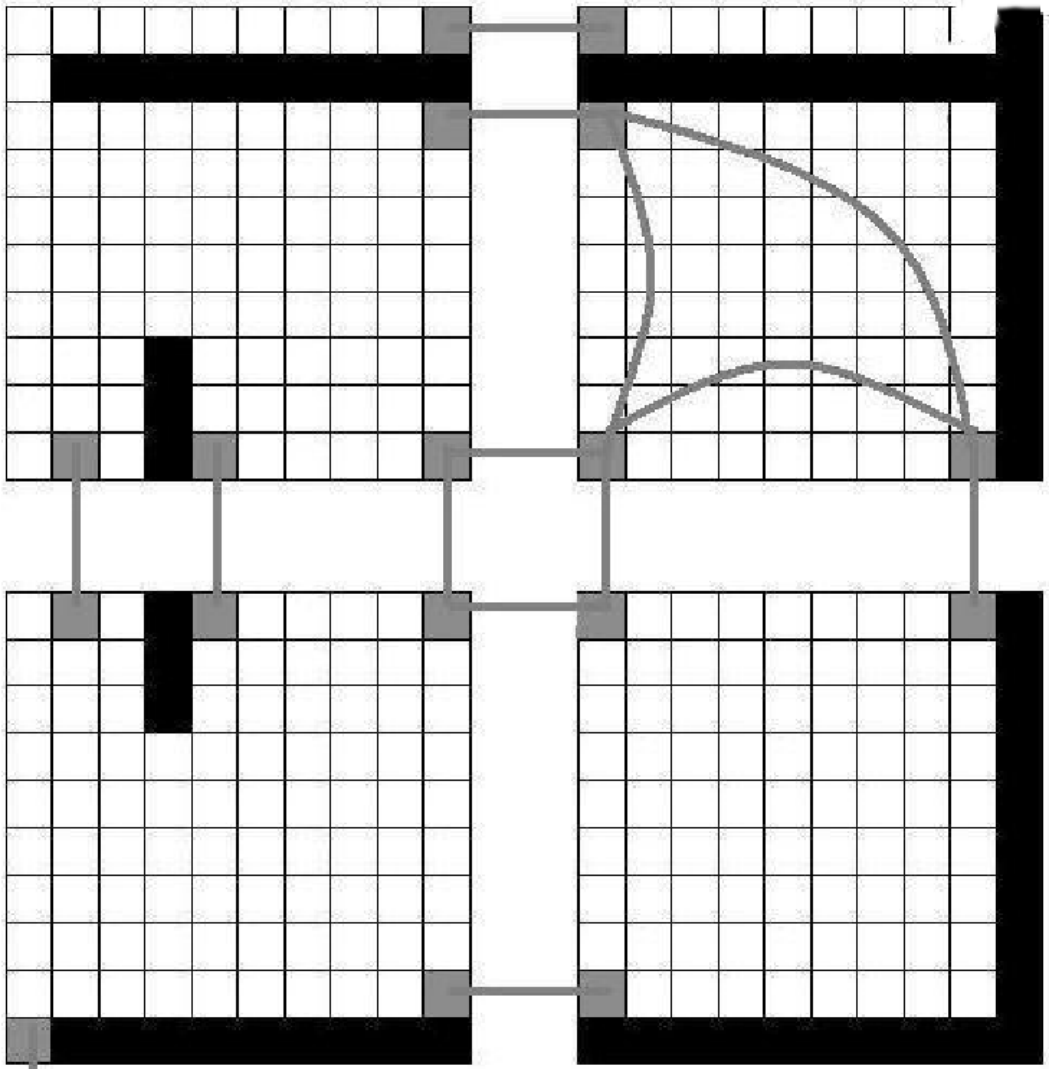
1. Superimpose sectors on top of tile grid
2. Build graph: one or more nodes per sector entrance (depending on width), intra-sector edges, inter-sector edges
3. Compute intra-sector distances (inter-sector distances = 1)

B. Find path in abstract graph

C. Smooth path

HPA* Preprocessing

Example: 2×2 sectors (most intra-sector connections omitted for clarity)



HPA*

```
// input: start location s, goal location g
// output: approx. of shortest path from s to g
// or failure
// (assumes pre-processed map)
function HPA*-FindPath(s,g):
  locate sectors containing s and g
  add intra-sector edges from s,g to entrances
  compute distances from s and g to respective
    entrances
  find abstract path from s to g by stitching
    intra-sectors paths together
  if none exists return failure
  smooth path
  return path
```

HPA* Performance

FAST without smoothing!

$\approx$ 10 times faster than A* on 512$\times$512 maps

But smoothing slows HPA* down a lot

After smoothing, path length is within 3% of optimal on average on game maps up to 512$\times$512

HPA* Improvements

The original HPA* implementation was improved in [3]:

- Using Dijkstra's algorithm (or UCS) for intra-sector distances
- On demand intra-sector computations
  (laziness pays off in dynamic environments!)
- Faster window-based smoothing

HPA* Evaluation

Advantages:

- Suitable for dynamic worlds: local terrain changes only trigger distance computation within a small number of sectors
- Simple implementation
- Can build multiple levels of abstraction

Disadvantages:

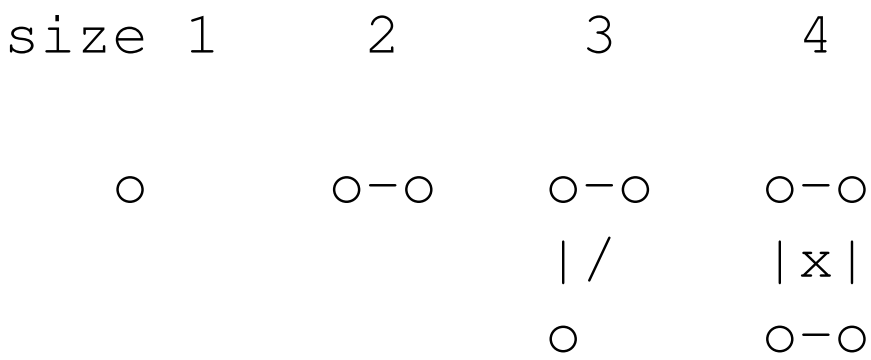
- Connecting s and g to the abstract graph can be time consuming
- Without time consuming smoothing path quality isn't great (often 20%+ longer than optimal)

Path Refinement A* (PRA*, [4])

Build abstraction hierarchy

We used “clique abstraction” on octile base-level grids

Clique = maximally connected subgraph



Map Abstraction

To abstract a map, scan for cliques of size 4,3,2 (say left to right, top down) and replace them by single nodes that are connected to their abstract neighbours

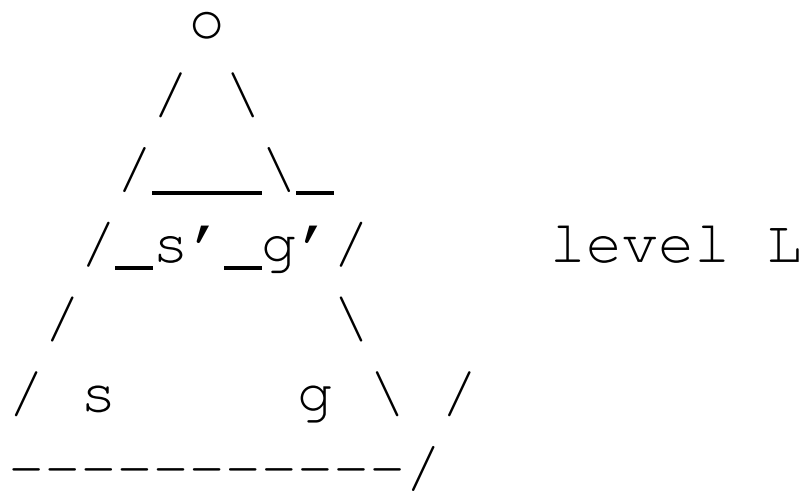
```
a-b-c-d-e-f
|x| |x|x|          A-B-C-D     A abstracts {a,b,g,h}
g-h i-j-k    ->    | |x|       B abstracts {c,d,i,j}
|x| |x|x|          E-F-G       C abstracts {e,k}
l-m-n-o-p-q                    D abstracts {f}     (orphan)
                               E abstracts {l,m}
                               F abstracts {n,o}
                               G abstracts {p,q}
```

Connect abstract nodes X, Y, if there is a path from each original node in X to an original node in Y. This conserves connectedness. Also, use centroids of abstracted nodes as location of abstract nodes

Iterate ...

Map Abstraction (continued)

Result: a collection of pyramids describing connected components of the original map



How to find paths?

- Pick an abstraction level L (halfway level is empirically good)
- In there, find abstract nodes s' and g' corresponding to nodes s, g
- Perform A* search from s' to g' in this level

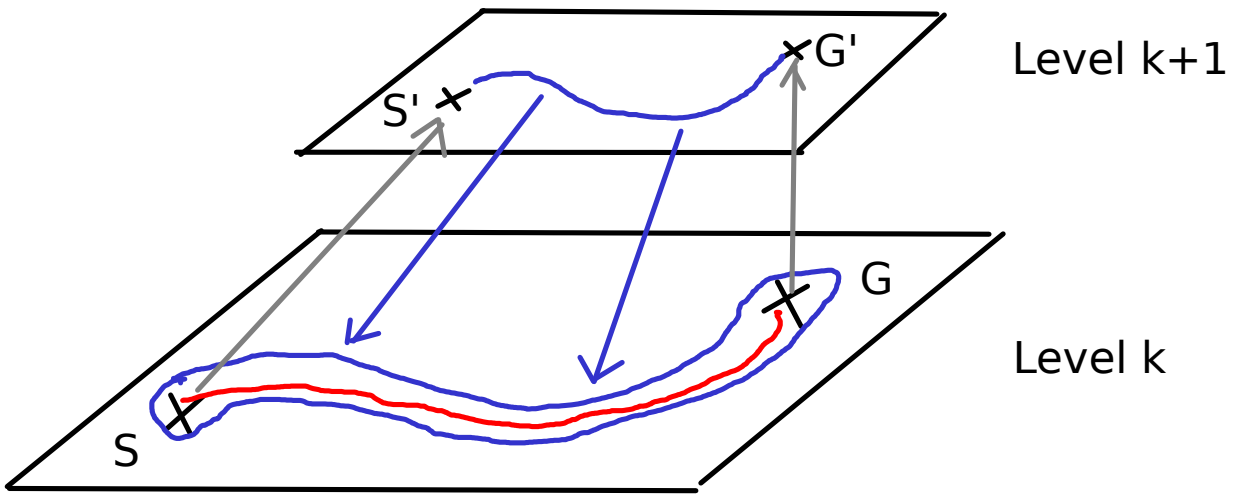
Path Refinement

Path refinement steps:

1. if base level not reached:
2. project found path down one level (also consider neighbours)
3. find optimal path in this corridor using A*
4. goto 1

For planar path finding applications one can use Euclidean distance on abstraction centroids (average coordinates of all abstracted nodes) as heuristic

Path Refinement Illustration



PRA* Performance

PRA* is FAST!

$\approx$ 10 times faster than A* on 512$\times$512 game maps

High-quality paths! 95% of the paths off by less than 3%

There is more:

- Interleaving path planning with path execution!
- After finding path at level l, just project the first k steps down
- Shorter corridor => faster
- But not much worse, because we know the global direction
- This is called PRA*(k)

PRA* Performance (continued)

Total amount of work bigger, but initial path planning operation very fast

- minimizes latency induced by planning operation
- objects can start moving right away
- saves time when object gets different move order, which happens often in video games

[AI-Lec 8 L13] Path Planning in Triangulations

Issues with grid-based path planning:

- Potentially crude approximation
  (what about circular object footprints or non-axis-aligned obstacles?)
- Objects occupy a whole tile
  (what about bigger objects?)
- Octile topology
  (what about any-angle motion?)

Geometry to the Rescue

Address these issues by considering free-space decompositions (a.k.a. tesselations):

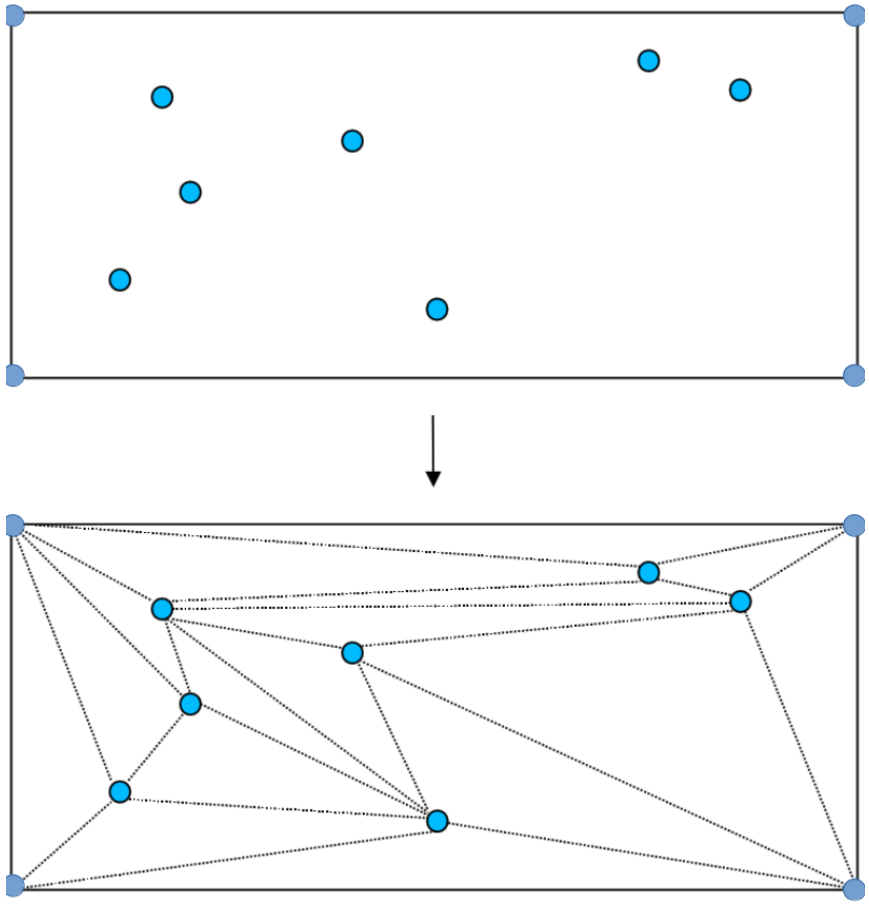
- quad-trees
- trapezoidal decomposition
- triangulations (a.k.a. navigation meshes)

Basic idea:

- decompose area around obstacles (free-space) into convex shapes
- for path planning first locate s and g, then hop from area to area
- smooth resulting path

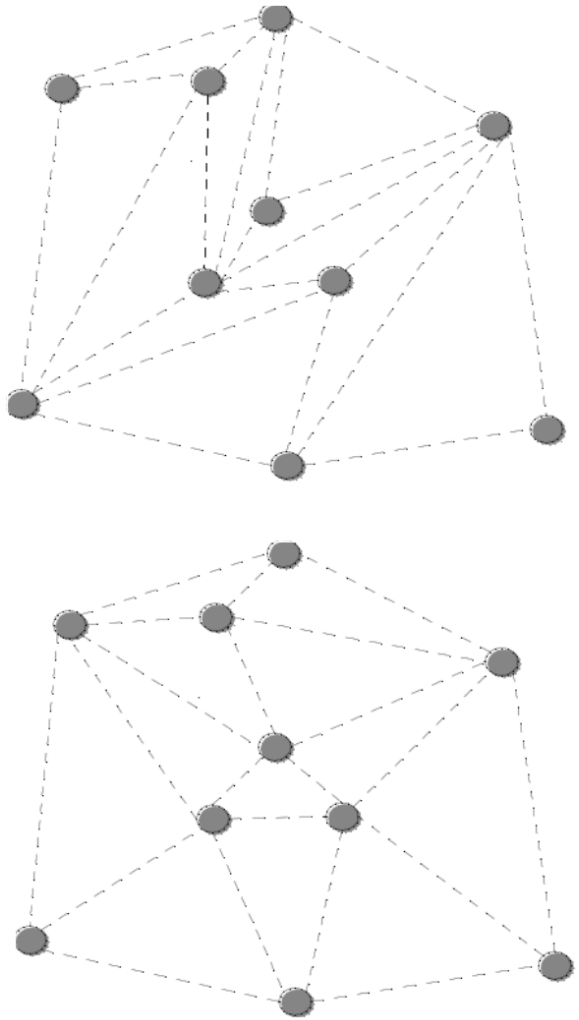
Triangulations

- Starting with an area (like a rectangle) and a collection of points (including the rectangle corners)
- Add edges between the points without such edges crossing
- Continue until no more such edges can be added
- The result is a triangle-based decomposition of the convex hull of the given points



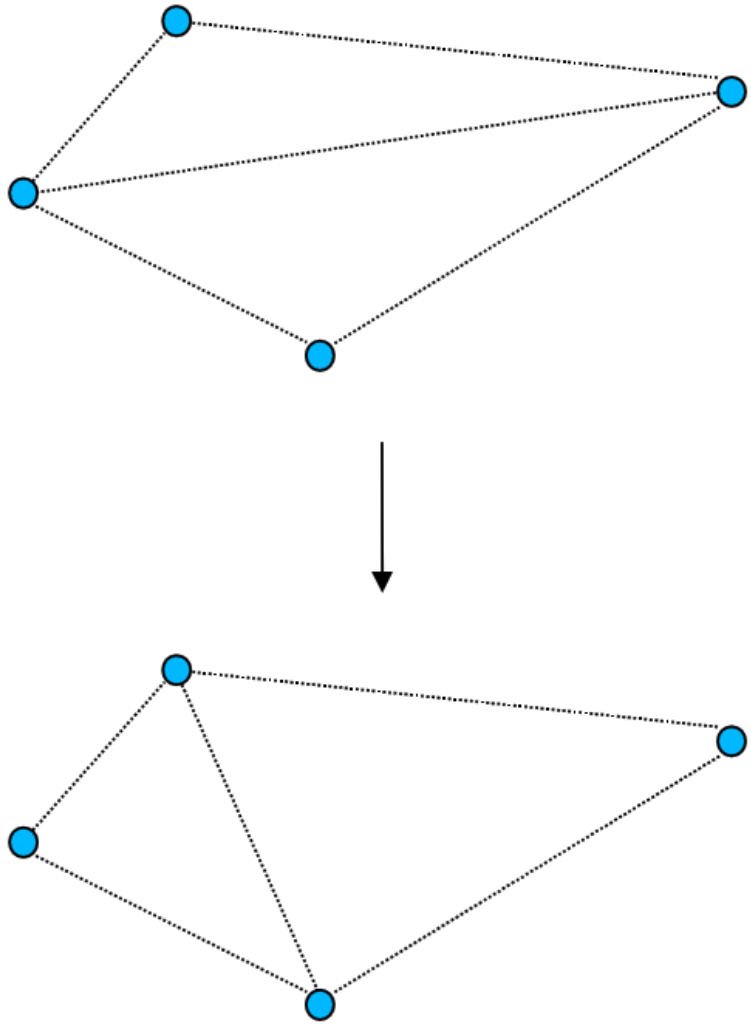
Triangulation Quality

- For a given point set many triangulations exist
- We would like to avoid sliver-like triangles which decrease locality and the quality of distance heuristics



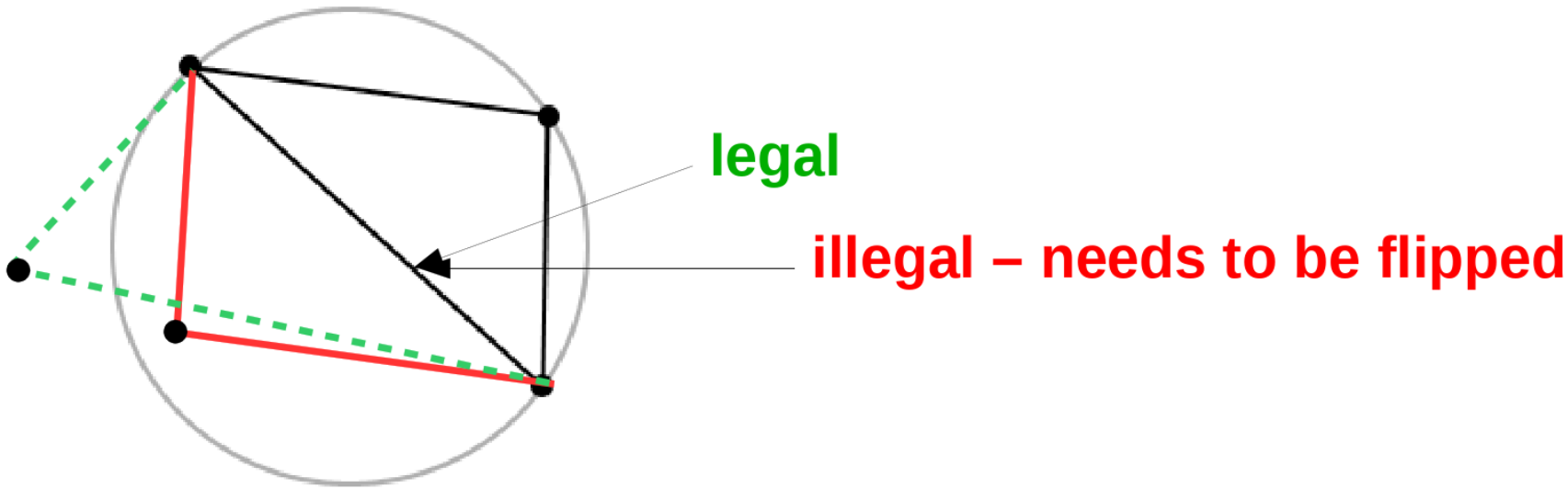
Delaunay Triangulations

- Triangulations which maximize the sorted interior angle vector (non-decreasing). I.e., the minimum angle is maximized
- Makes “nice” triangulation: tends to avoid thin, sliver-like triangles
- Can be done locally by “edge flipping” diagonals across quadrilaterals



Delaunay Triangulation Characterization

A triangulation maximizes the minimal angle iff for each quadrangle the circumcircle of each triangle does not contain the fourth point



Details including a correctness proof of the determinant-based in-circle test (seen in Lab 08) can be found in [12]

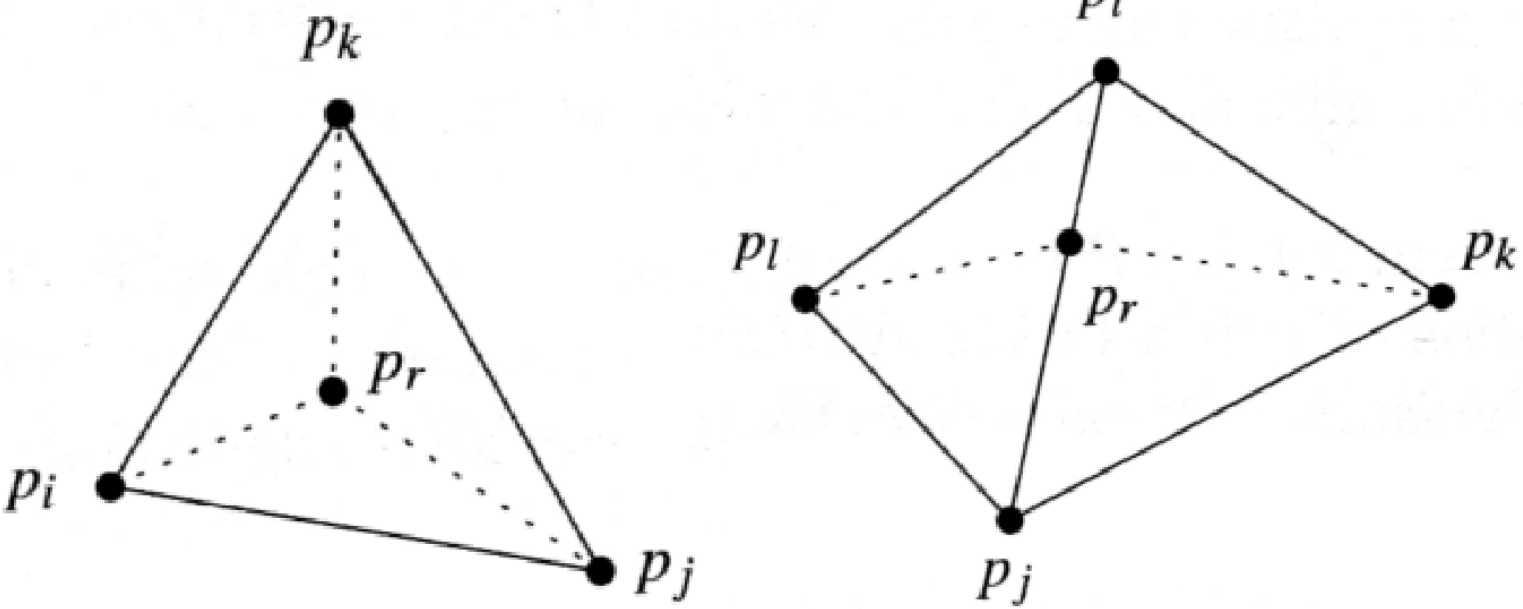
Computing Delaunay Triangulations

1. Initialize triangulation T with a "big enough" helper bounding triangle that contains all points of P
2. Randomly choose a point p_r from P
3. Find the triangle Δ that p_r lies in
4. Subdivide Δ into smaller triangles that have p_r as a vertex
5. Flip edges until all edges are legal
6. Repeat steps 2-5 until all points have been added to T

Randomized algorithm. Expected runtime $\Theta(n \log n)$

Can also be computed using Divide & Conquer

Inductive Step



Point Localization in Triangulations

Triangles are given as a vector. They store point coordinates and indexes of neighbouring triangles

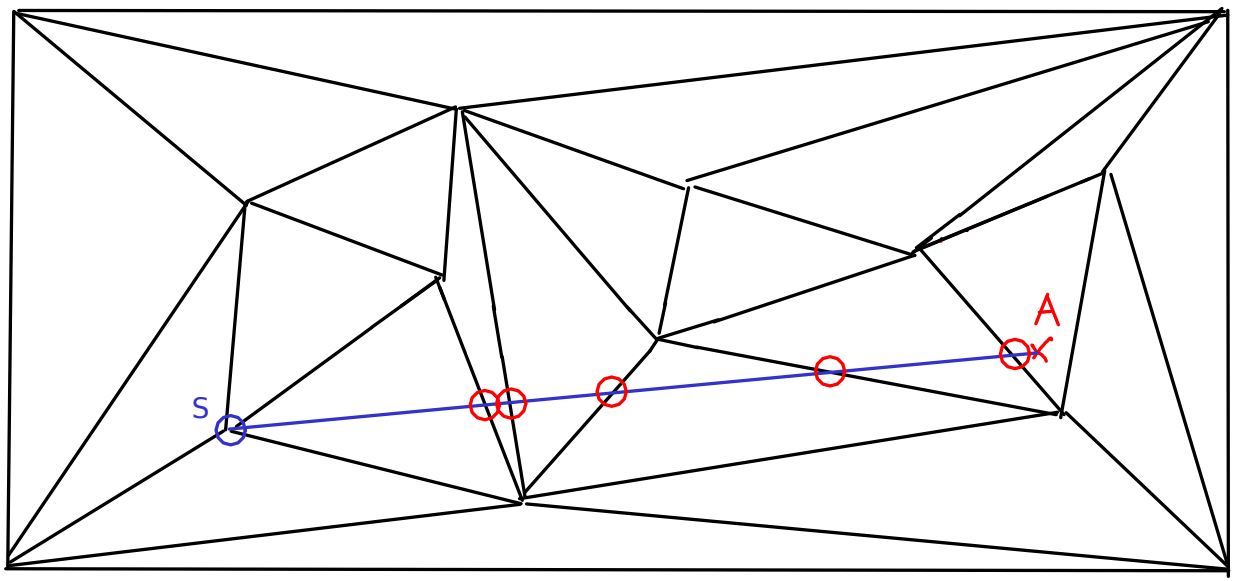
Task: For a given point (x, y) find the triangle(s) it resides in

- Brute force [$\Theta(n)$ runtime for n triangles]
  [How to check whether a point lies inside a triangle? (exercise)]
- By maintaining a decision tree while contructing the triangulation that identifies triangles based on x, y questions
  [expected runtime $\Theta(\log n)$, see [5] for details]

```
      x < y?              - Depth logarithmic in n
    +/      \-              (on average)
  y > 2?    y < 4?        - Update subtrees when adding
+/   \-   +/   \-           nodes to triangulation
Δ1   Δ2   Δ3   Δ4          - Tricky to implement
```

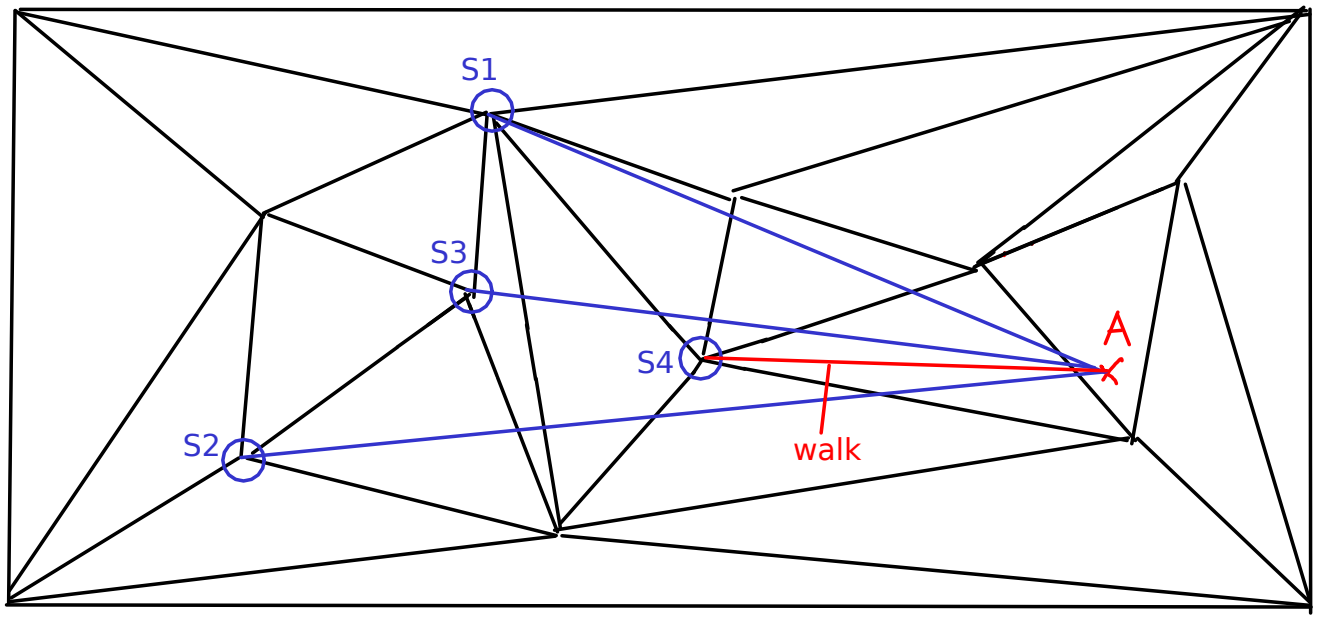
Straight Walk Algorithm

- Start with random triangle. Then walk towards goal location A by crossing edges that intersect the line between A and the opposite initial triangle point
- This is called a Straight Walk [11]. Its average case runtime is $\Theta(\sqrt{n})$ in homogenous triangulations consisting of n triangles
- What is its worst-case runtime?



Jump and Walk Algorithm

Idea: sample k triangles and pick the closest to goal point A as starting point



Expected runtime: time for sampling + expected time for walking

Minimal when both terms are asymptotically equal: sweetspot is $\Theta(\sqrt[3]{n})$ [10]. For $n = 1,000,000$, $\log_2 n \approx 20$, whereas $\sqrt[3]{n} = 100$. So, Jump and Walk point localization is competitive with tree-based point localization in practice, and much simpler to implement

Sector-Based Jump and Walk Method

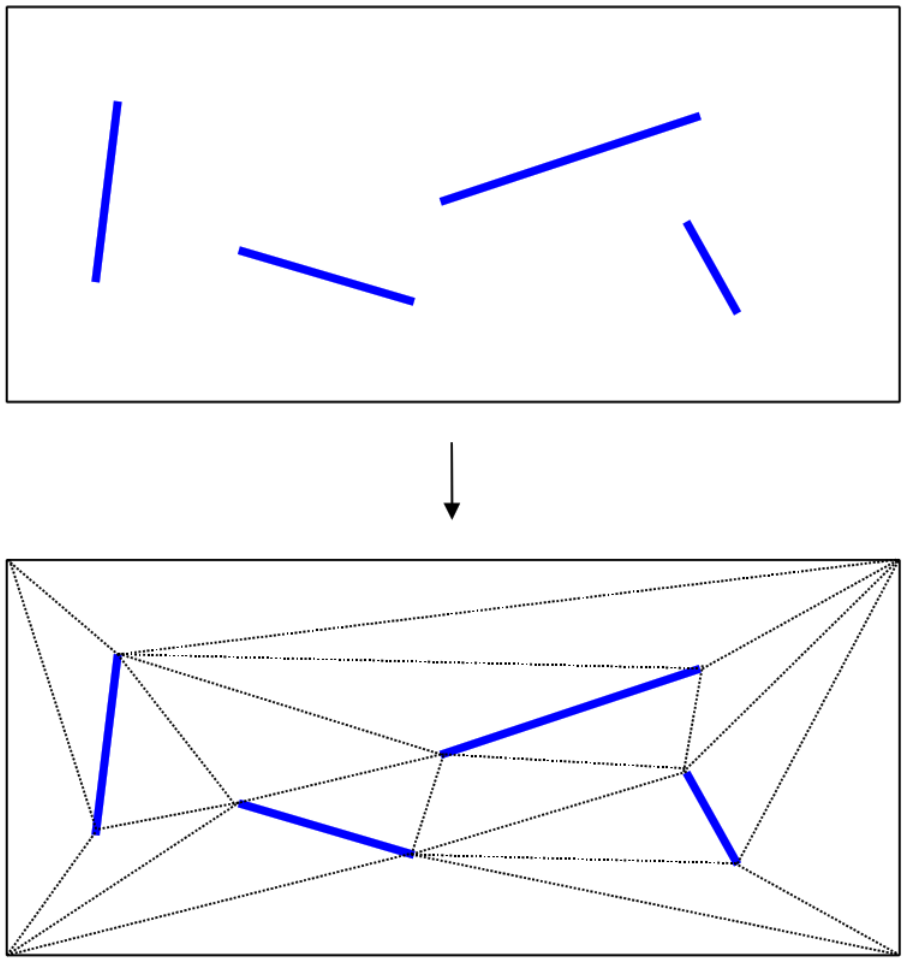
Even better: use sectors + Jump and Walk (approaching $O(1)$, see [9])

Main steps:

- preprocessing: superimpose sectors (similar to HPA*)
- maintain starting triangle for each sector (could be void at first)
- to locate (x, y) compute sector and start walking at sector start triangle
- if none exists, use original Jump and Walk method
- store found triangle in sector

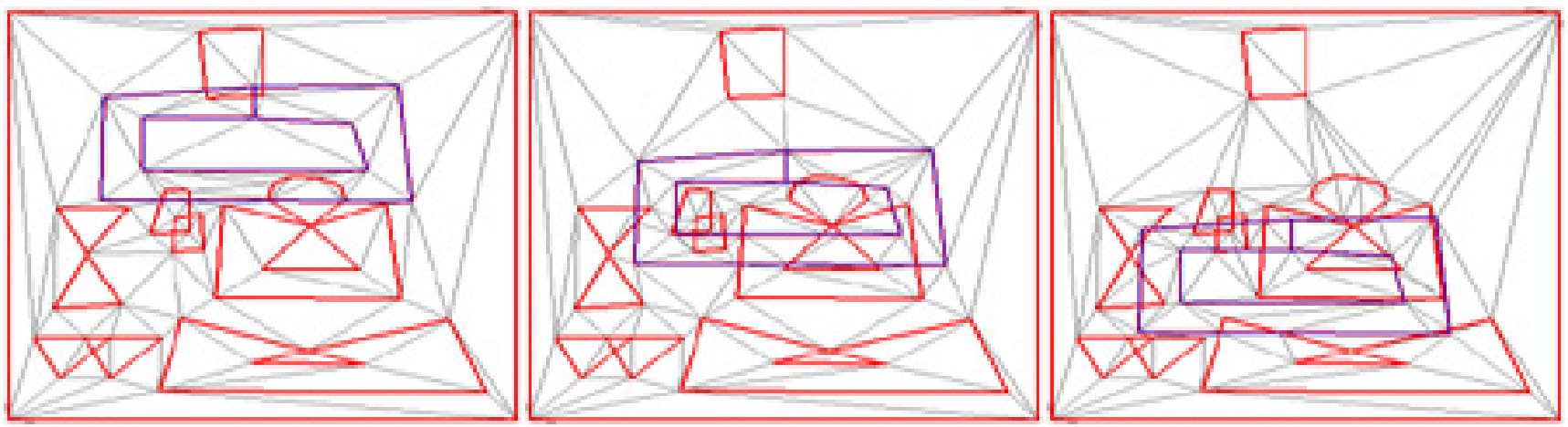
Constrained Triangulations

- Triangulations where certain (constrained) edges are required to be in the triangulation
- Then other (unconstrained) edges are added as before
- Constrained Delaunay Triangulations maximize the minimum angle while keeping constrained edges
- Above algorithm can be used with modifications



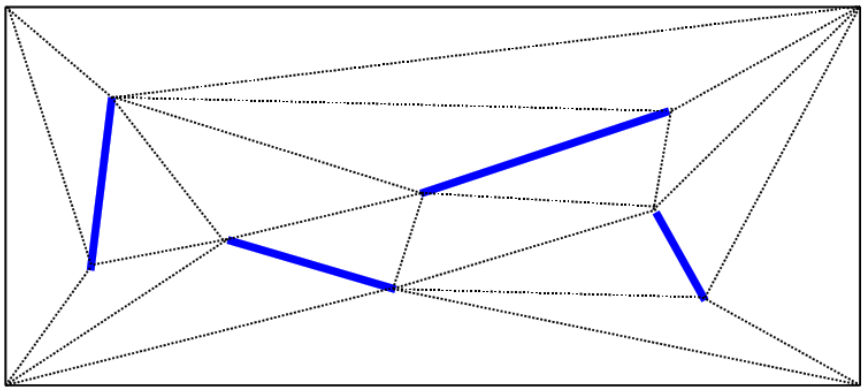
Dynamic Constrained Delaunay Triangulations (DCDT)

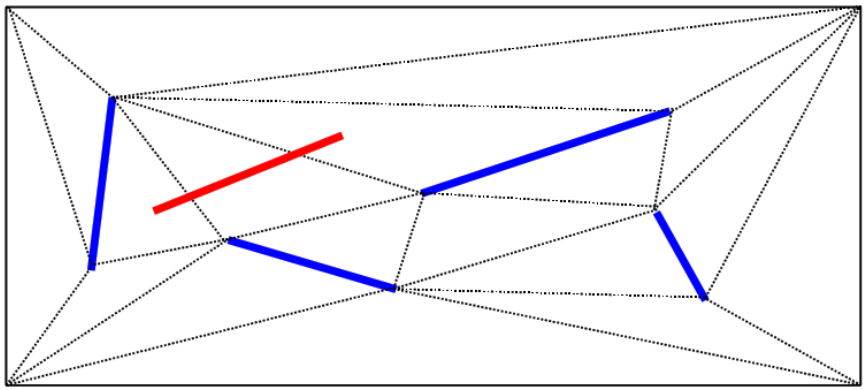
- Marcelo Kallmann's DCDT software [6,7] can repair a triangulation dynamically when constraints change
- Repairs can be made using local information allowing it to work in a real-time setting

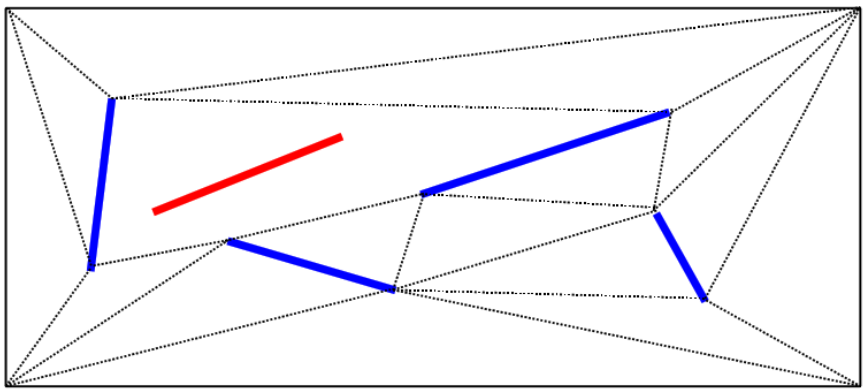


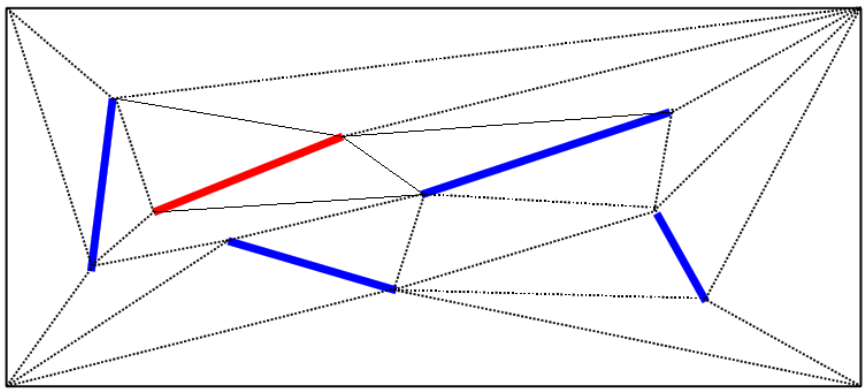
[Shown video]

Example: Add Constraint Segment



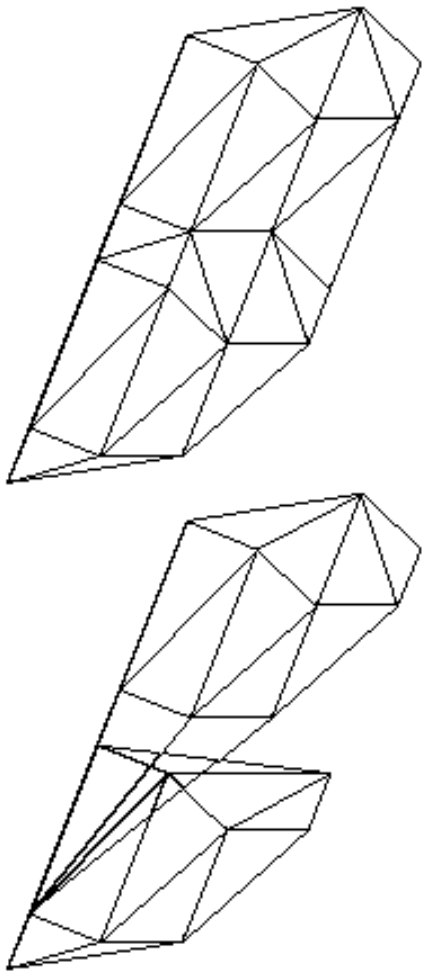






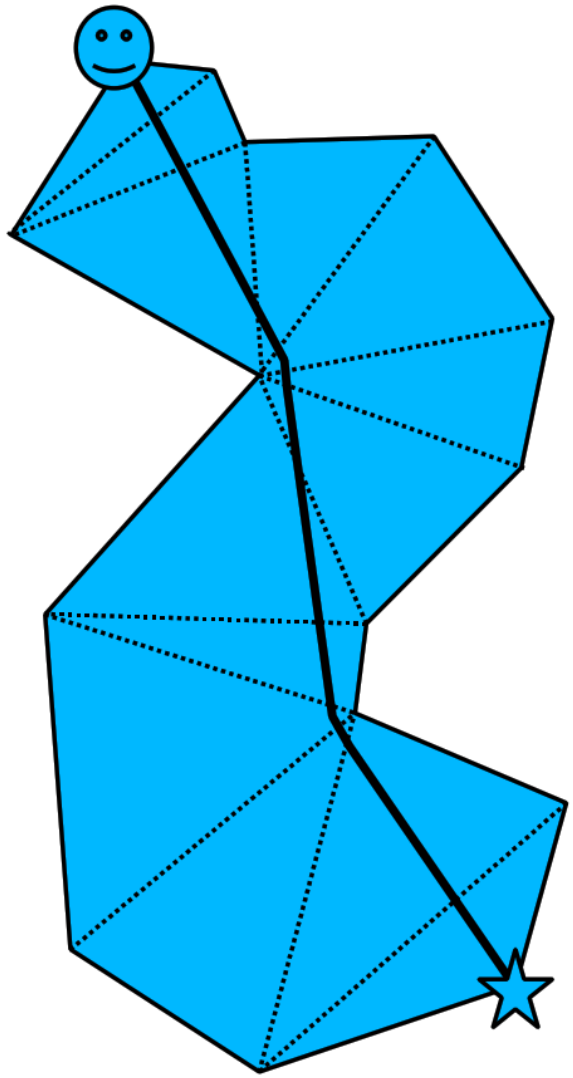
Robustness of Geometric Computations

- Using fixed-length floating point arithmetic can cause geometric algorithms
  - to crash
  - to hang
  - to produce incorrect output
- Kallmann's DCDT software suffers from this in rare cases
- We developed a library for DCDT using exact rational and interval arithmetic (soon to be on GitHub, [9])



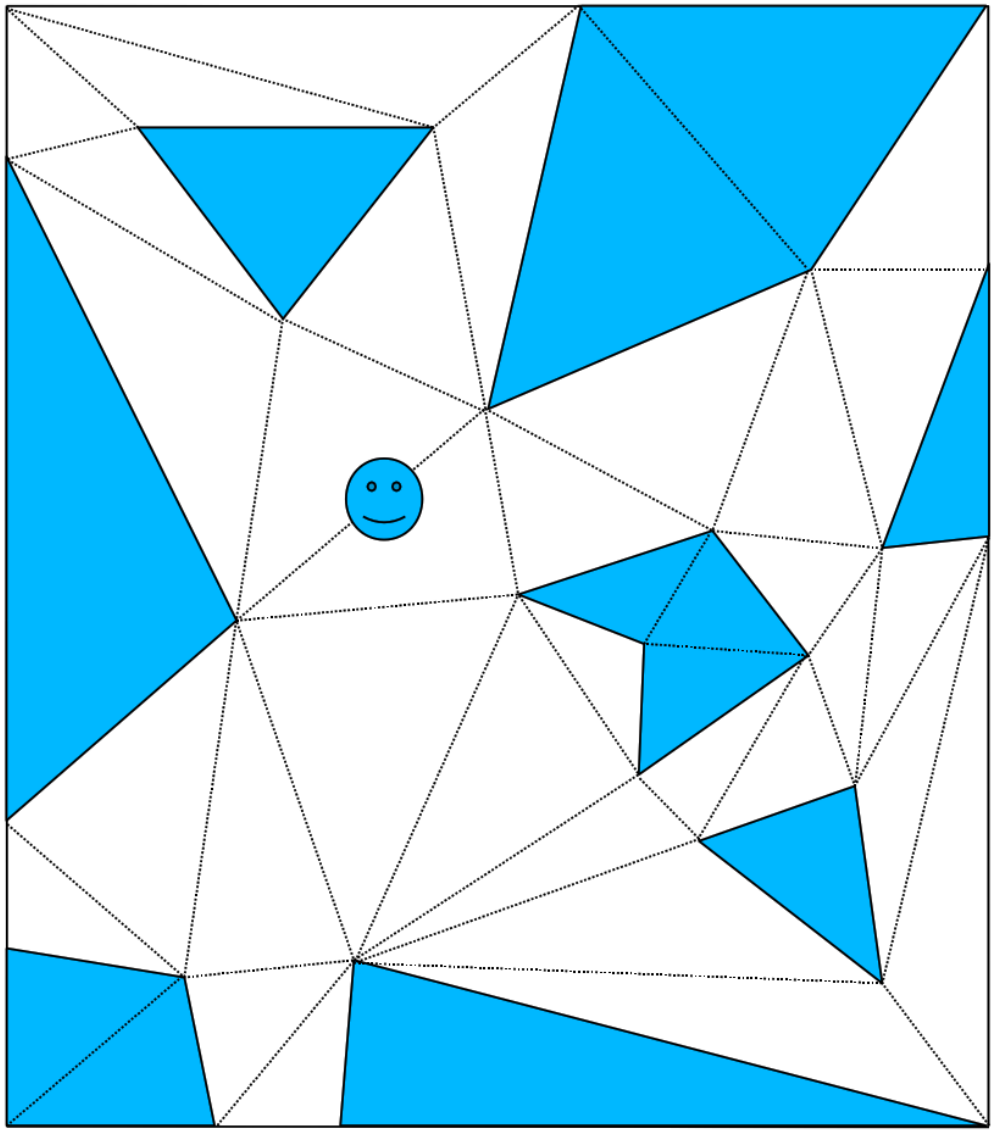
[AI-Lec 9 L15] Triangulation-Based Path Planning [8]

- Using a constrained triangulation with barriers represented as constraints
- Find which triangle the start (and goal) point is in Search adjacent triangles across unconstrained edges
- Finds a channel of triangles inside which we can easily determine the shortest path



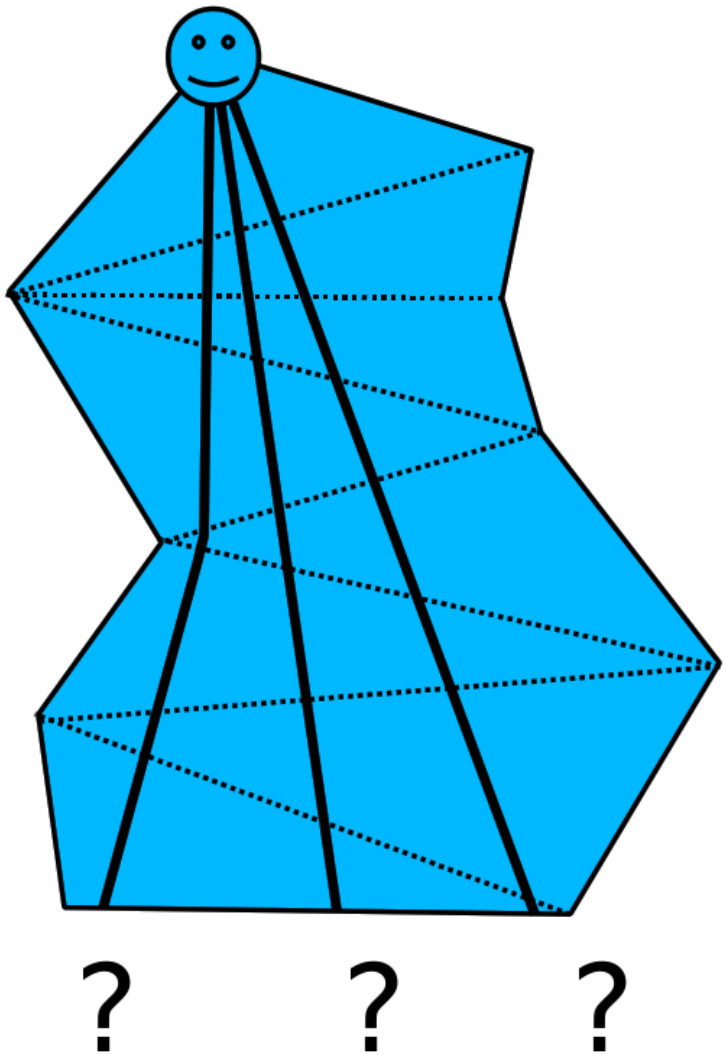
Triangulation-Based Path Planning: Advantages

- Remedies grid-based methods' deficiency with off-axis barriers
- Representing detailed areas better doesn't complicate "open" areas
- Triangulations have much fewer cells and are more accurate than grids
- Can deal with non-point objects quite easily (below)



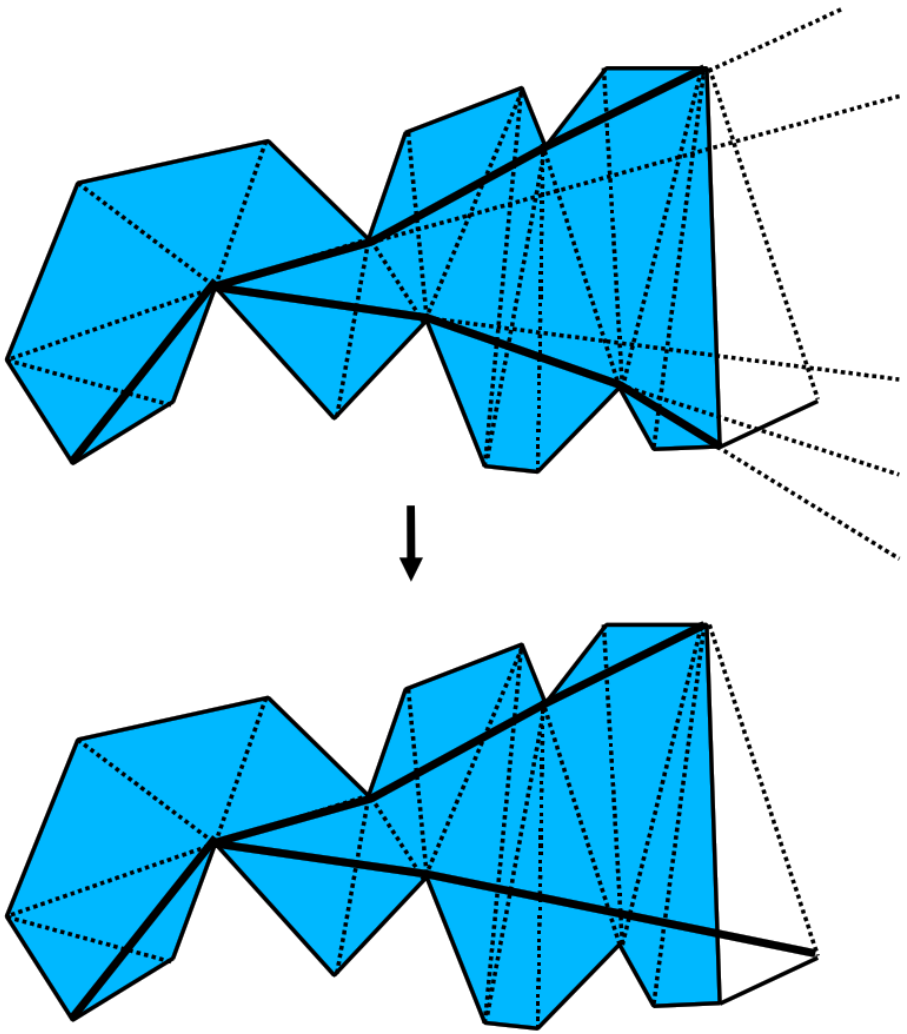
Triangulation-Based Path Planning: Disadvantages

- Curved obstacles must be approximated by straight segments
- We do not know what path we will take through the triangles until after we have found the goal
- This can lead to either suboptimal paths or multiple paths to triangles



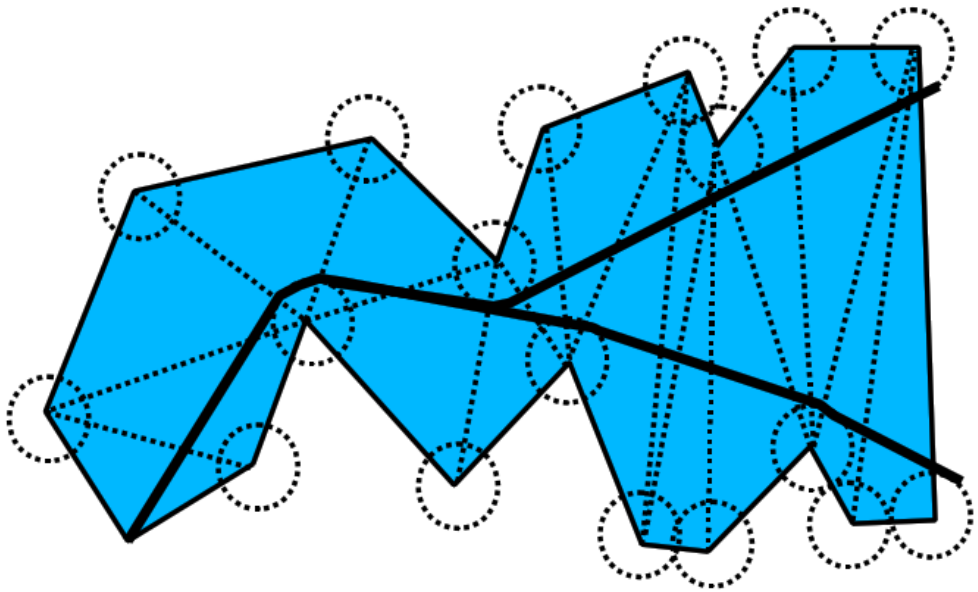
Funnel Algorithm [8 p.52]

- To find the exact path through a channel of triangles, we use the funnel algorithm
- It finds the shortest path in the simple polygon in time linear in the number of triangles in it
- Maintains a funnel which contains the shortest path to the channel endpoints so far
- Funnel is updated for each new vertex in the channel by finding the wedge in which the new endpoint lies



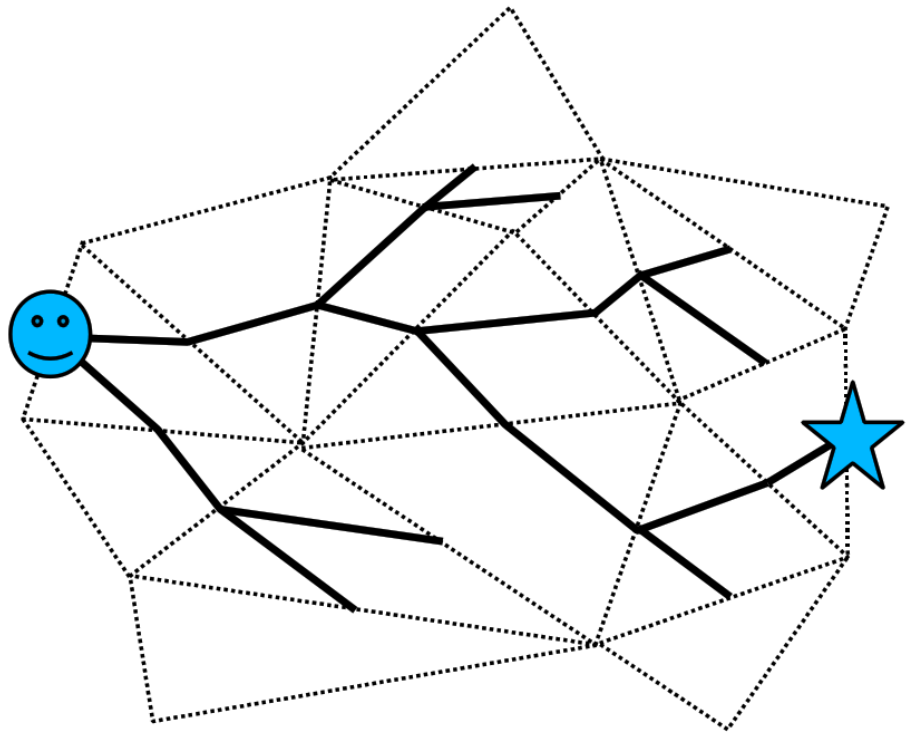
Modified Funnel Algorithm

- For circular objects with non-zero radius
- Conceptually attaches circles of equal radius around each vertex of the channel
- Considers segments tangent to these circles and arcs along them



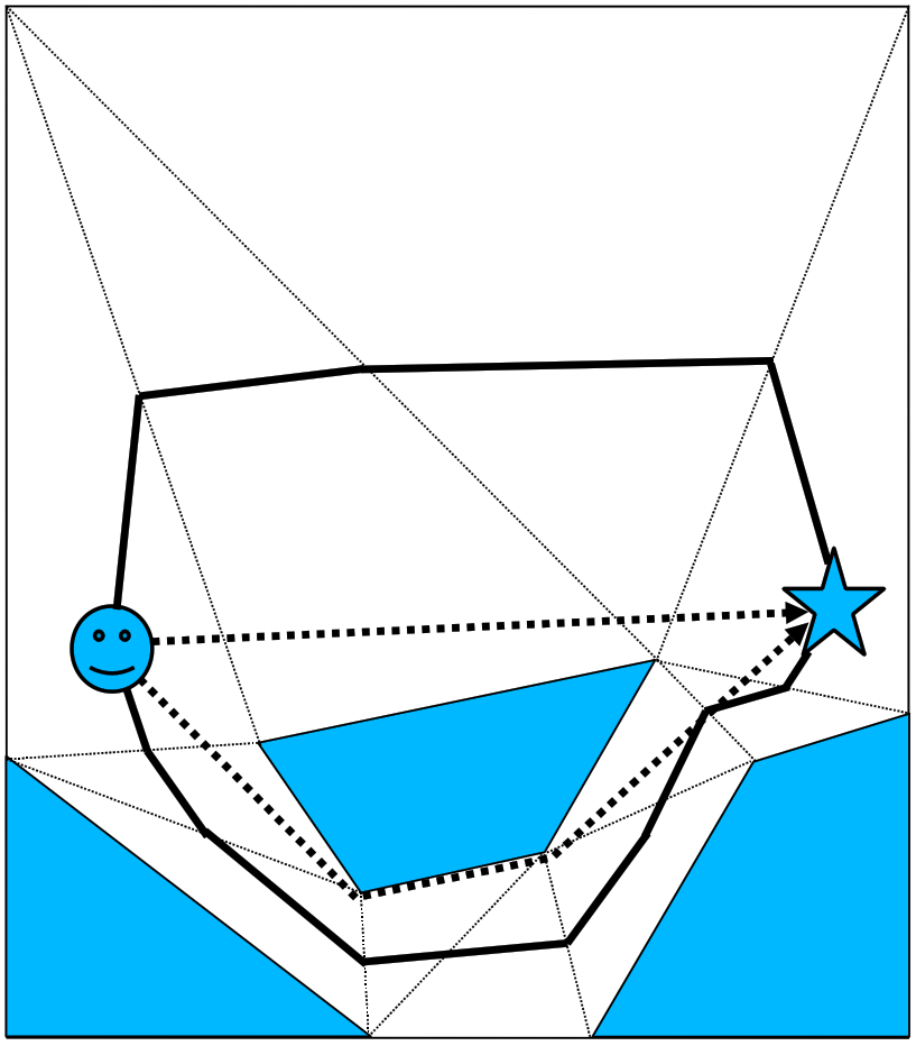
Naive Search in Triangulations

- Assume straight-segment paths between edge mid-points
- Run A* from start triangle looking for the goal triangle
- *g* cost: sum of straight-segment lengths on path
- *h* cost: Euclidean distance between current point and goal location (consistent)



Naive Search: Pros and Cons

- Considers each triangle once and has fairly good distance measures
- So, finds paths quickly
- However, in cases like the example on the right, it thinks a path through the bottom channel is shorter than one through the top
- So it may result in suboptimal paths



How to Find Optimal Paths?

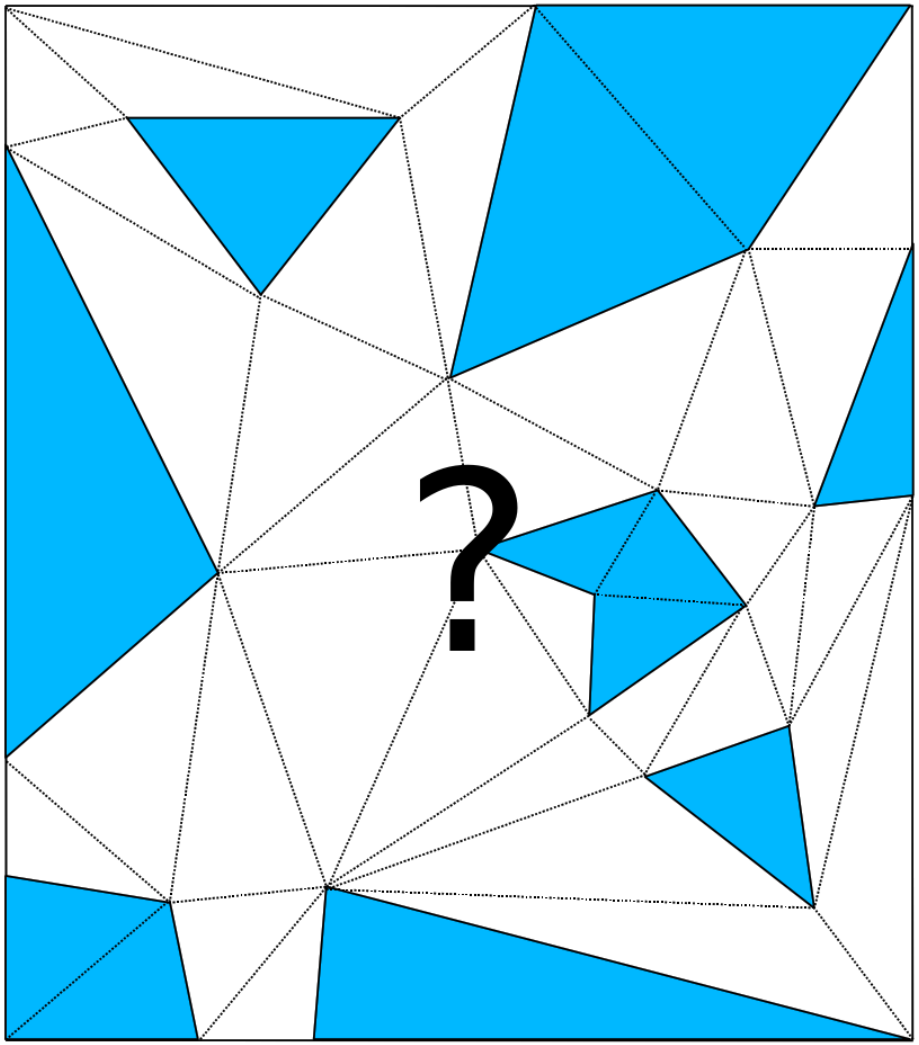
- (Under)estimate the distance travelled so far (e.g., take minimum distance of reaching any point in current triangle)
- Allow multiple paths to any triangle
- When a channel is found to the goal, calculate the length of the shortest path in this channel using the (modified) funnel algorithm
- If it is the shortest path found so far, keep it;
  otherwise, reject it (anytime algorithm)
- When the lower bound on the distance travelled so far for the paths yet to be searched exceeds the length of the shortest path, the algorithm ends and we have found an optimal path

Triangulation A* (TA*)

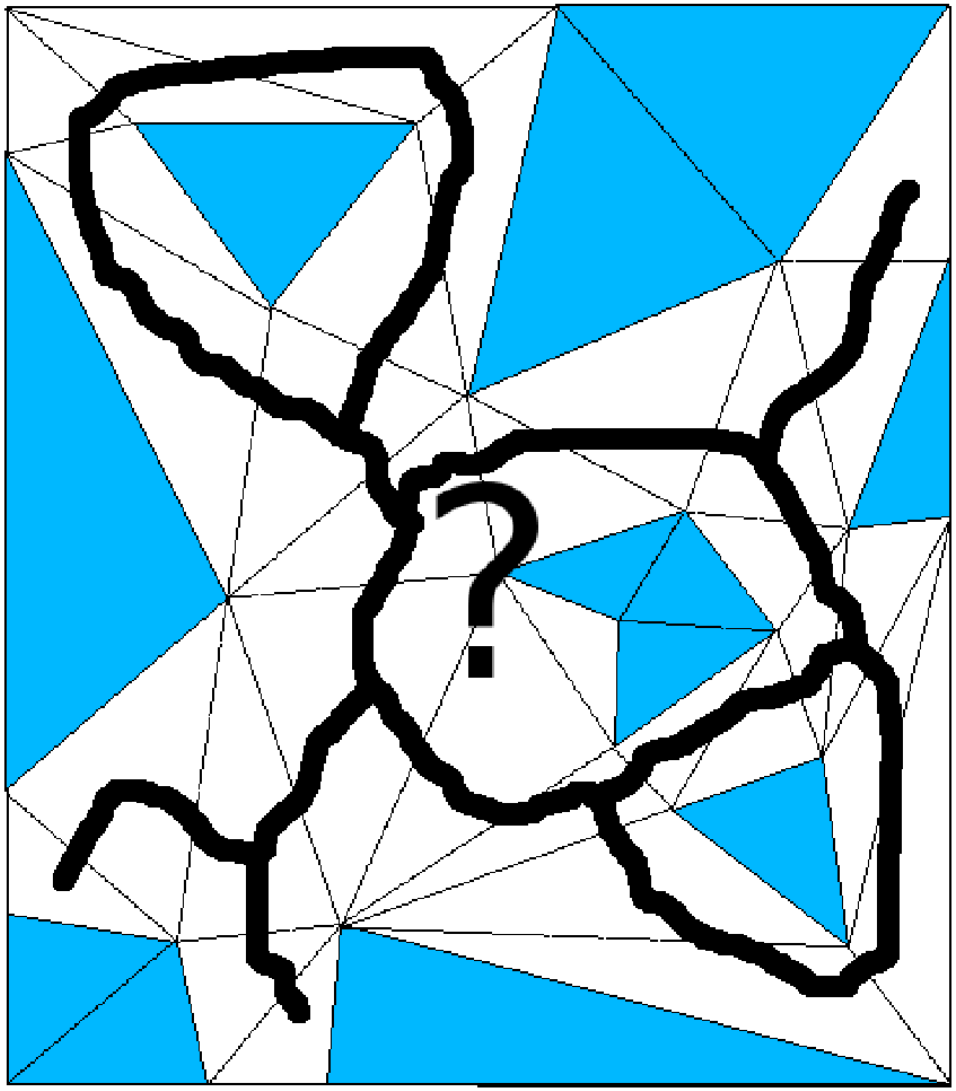
- Search running on the base triangulation
- Uses a triangle for a search state and the adjacent triangles across unconstrained edges as neighbours
- Using anytime algorithm and considering multiple paths to a triangle as described earlier
- For a heuristic (h-value), take the Euclidean distance between the goal and closest point on the triangle's entry edge
- Calculate an underestimate for the distance-travelled-so-far (*g*-value) (details: [8] p.67)
- Only consider triangles once until the first path is found
- Continue searching until time runs out or lower bound meets or exceeds current shortest distance

Triangulation Reduction

- Want to reduce the triangulation without losing its topological structure
- Determine triangles as being decision points, on corridors, or in dead ends
- Map a triangle to a degree-*n* node when it has exactly *n* triangles adjacent across unconstrained edges
- After mapping, collapse degree-2 corridors

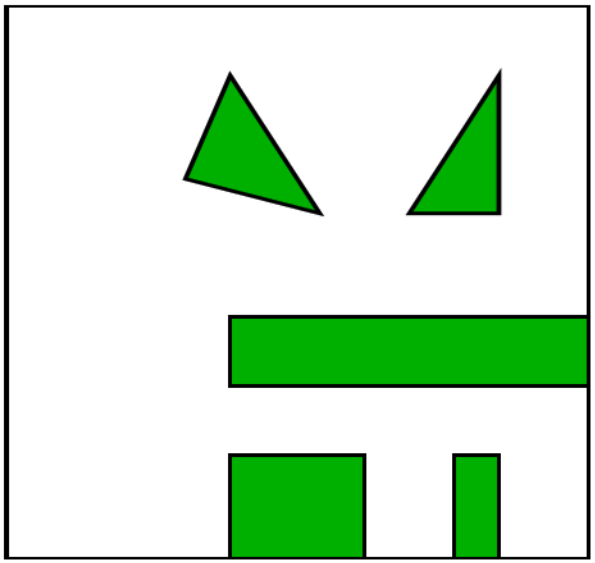


Topological View

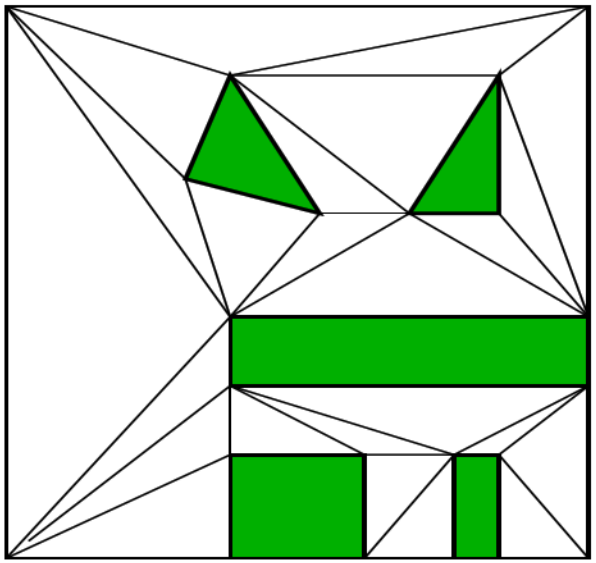


Triangulation Reduction Process

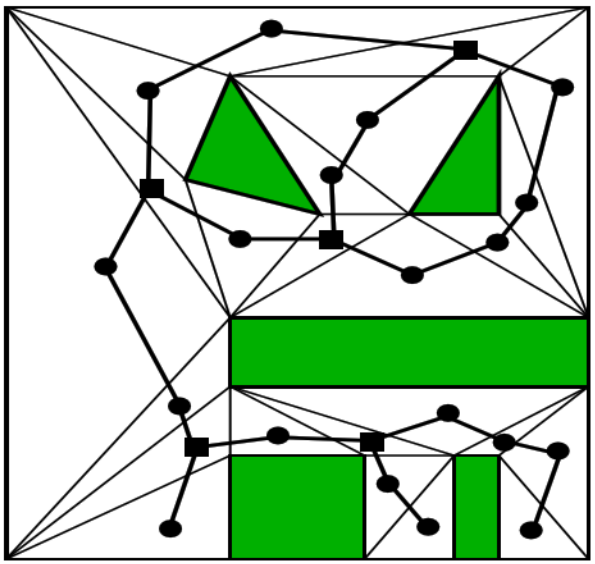
a) Polygon World



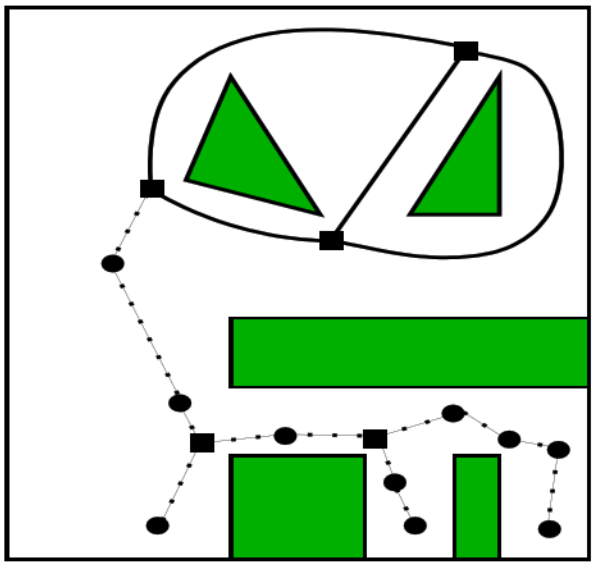
b) Triangulated World



c) Triangle Graph

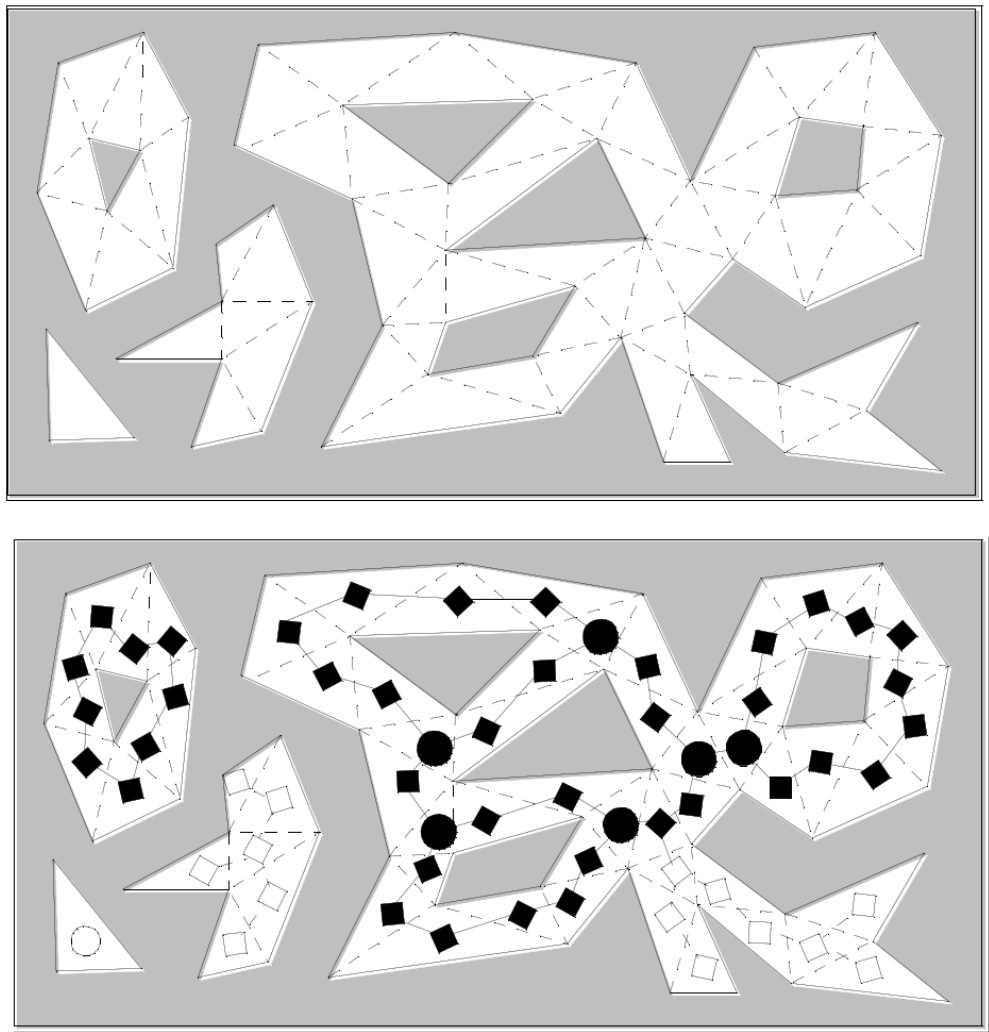


d) Abstract Triangle Graph

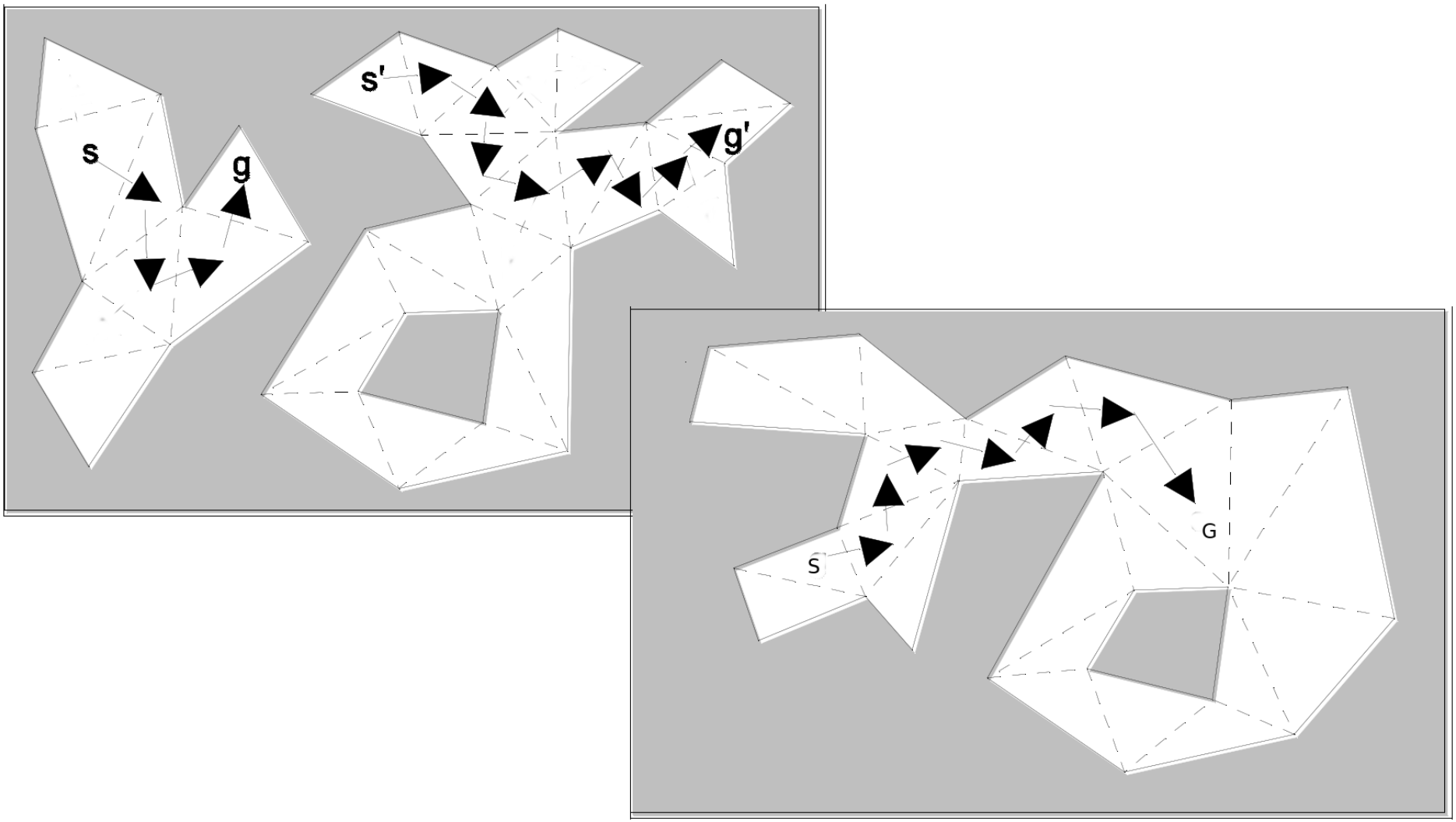


Reduction Example

- Path planning in tree components (degree-1, empty squares) and corridors (degree-2, solid squares) is easy
- The only real choice points are degree-3 triangles (solid circles)
- After corridor compression the resulting search graph has size linear in the number of islands!

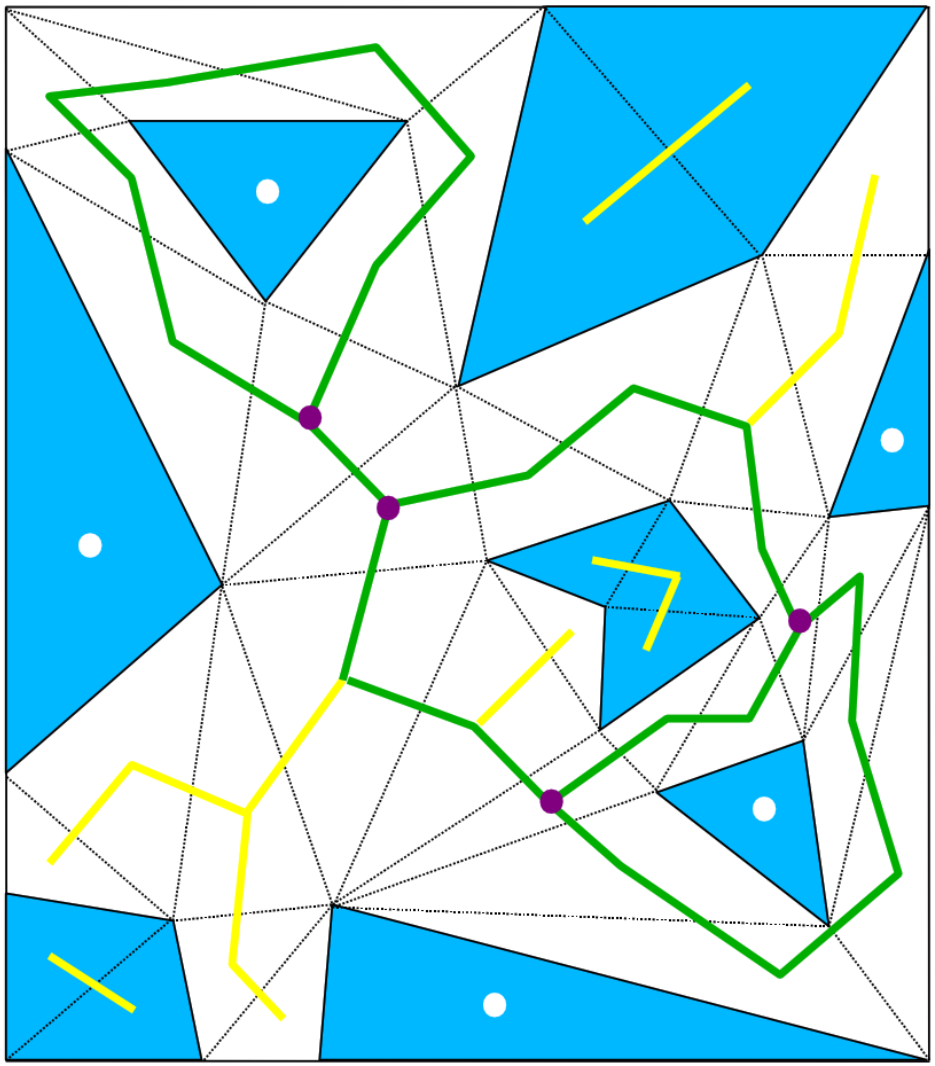


Simple Special Cases – No Search Required



Abstraction Information

- Adjacent structures
- Choke points (the narrowest point between this triangle and the adjacent structure)
- A lower bound on the distance to each adjacent structure
- Triangle “width”
- Using this graph we can find paths for differently sized objects



Triangulation Reduction A* (TRA*)

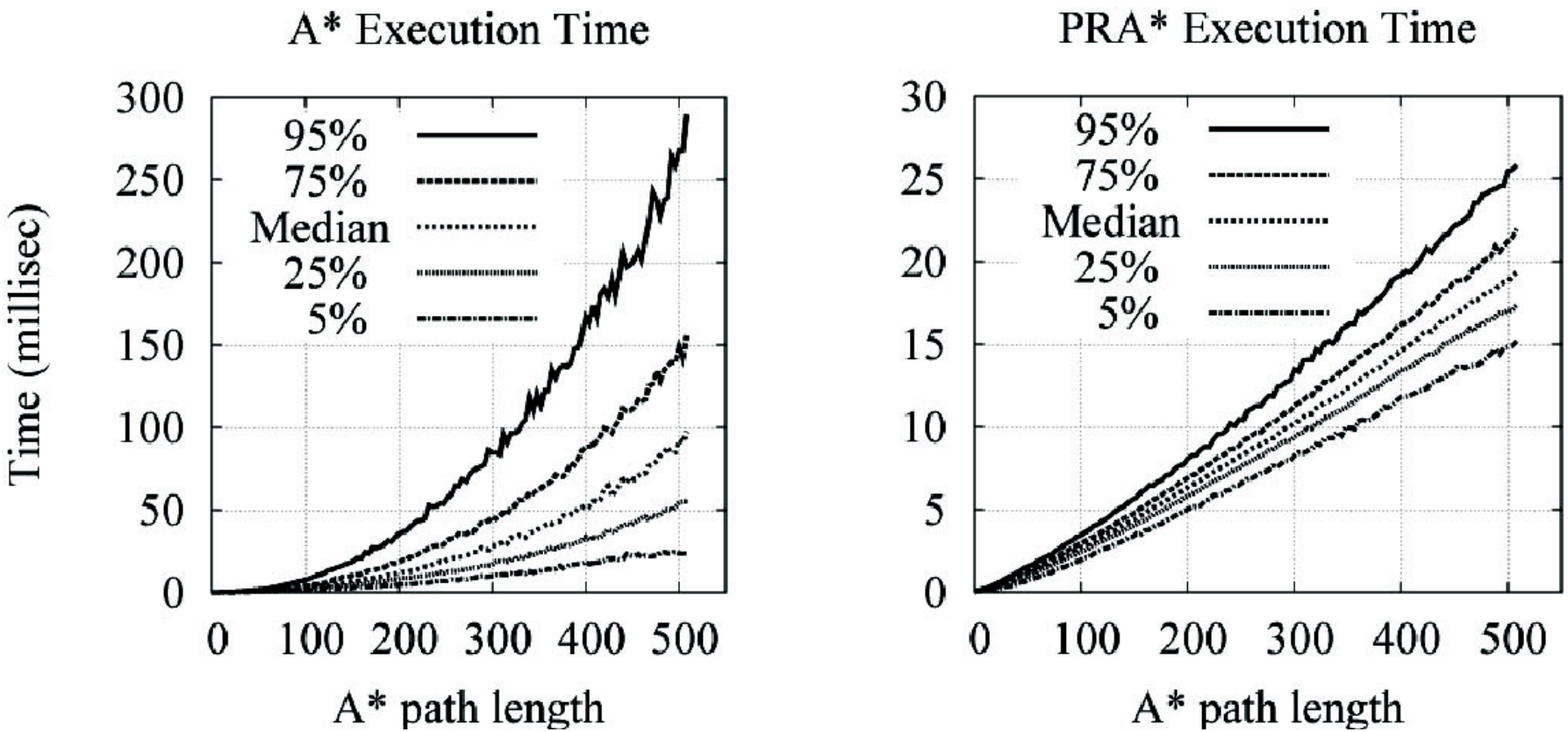
- TA* running on the abstraction we just described
- First check for a number of "special cases" in which no actual search needs to be done
- Move from the start and goal to their adjacent degree-3 nodes
- Use degree-3 nodes as search states and generate their children as the degree-3 nodes adjacent across corridors
- As with TA*, use an anytime algorithm, allowing multiple paths to a node, and use the same g- and h-values

Experimental Setup

- 116 maps scaled to 512 $\times$ 512 tiles:
  - 75 Baldur's Gate maps (grid of tiles marked traversible or untraversible)
  - 41 WarCraft III maps (grid of types of terrain and heights where paths cannot cross height differences without ramps or boundaries between different types of terrain)
  - 8-connected grids
- 1280 paths in each, with A* length between 0 and 511 and categorized into one of 128 buckets based on length
- Compared TA* and TRA* to A* and PRA* using the same maps and paths

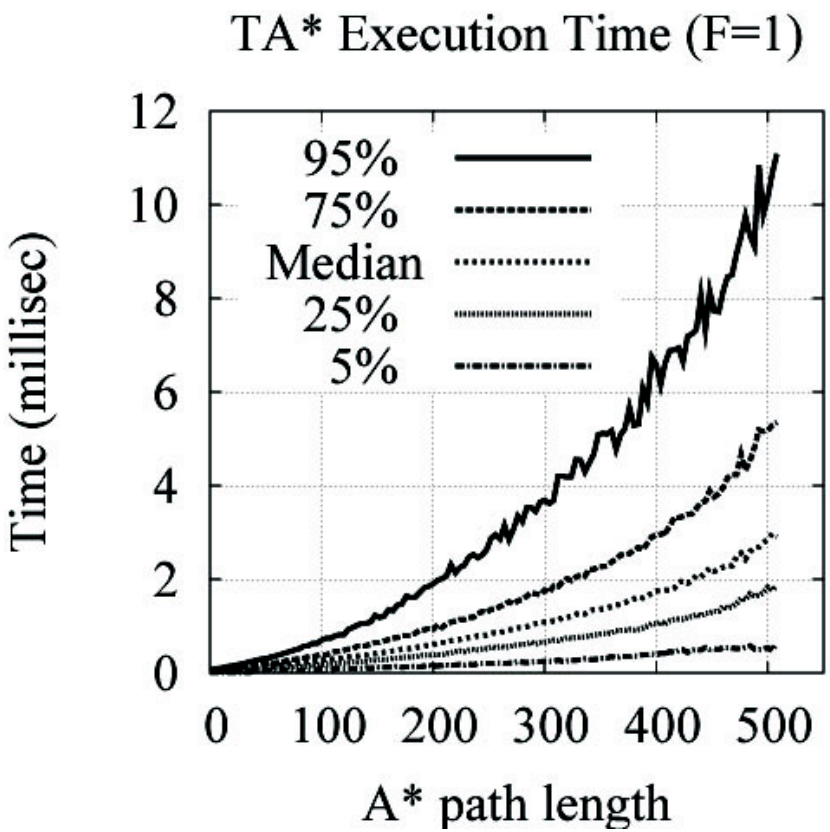
Execution Times A* vs. PRA*

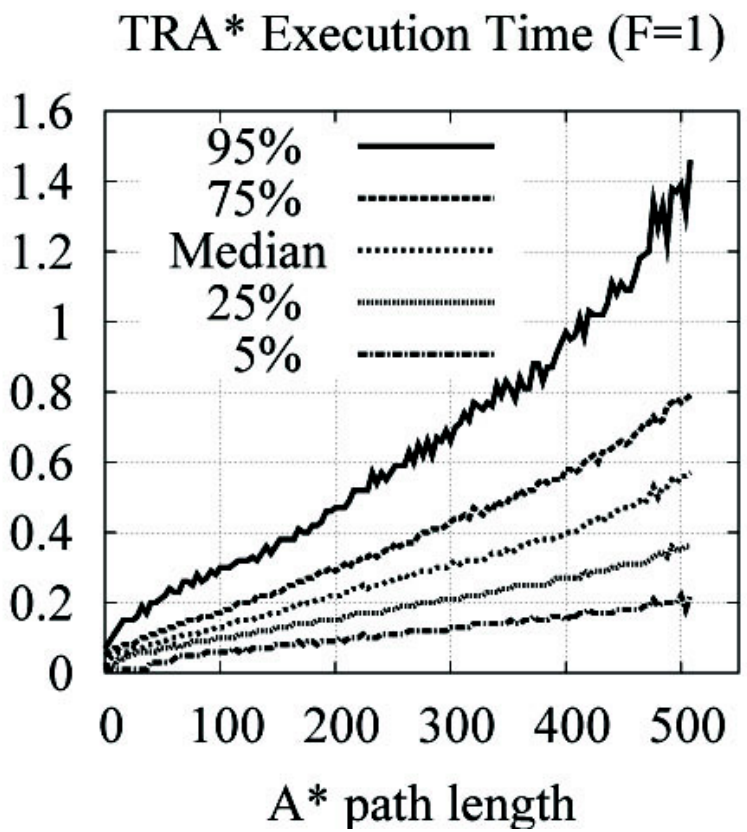
Halfway abstraction layer chosen for PRA*



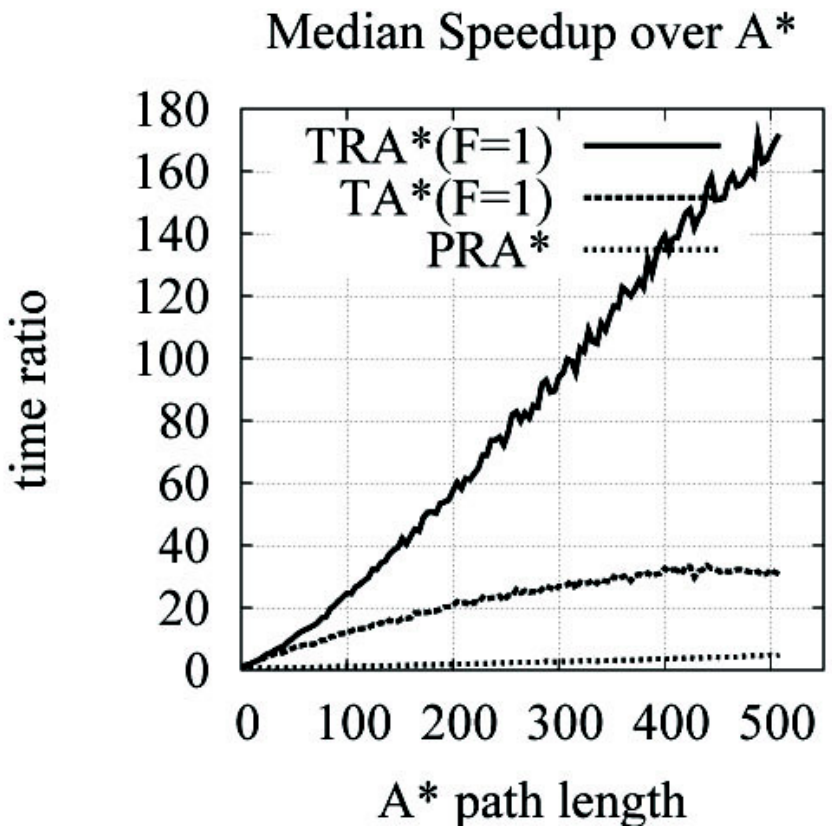
Execution Times TA* vs. TRA*

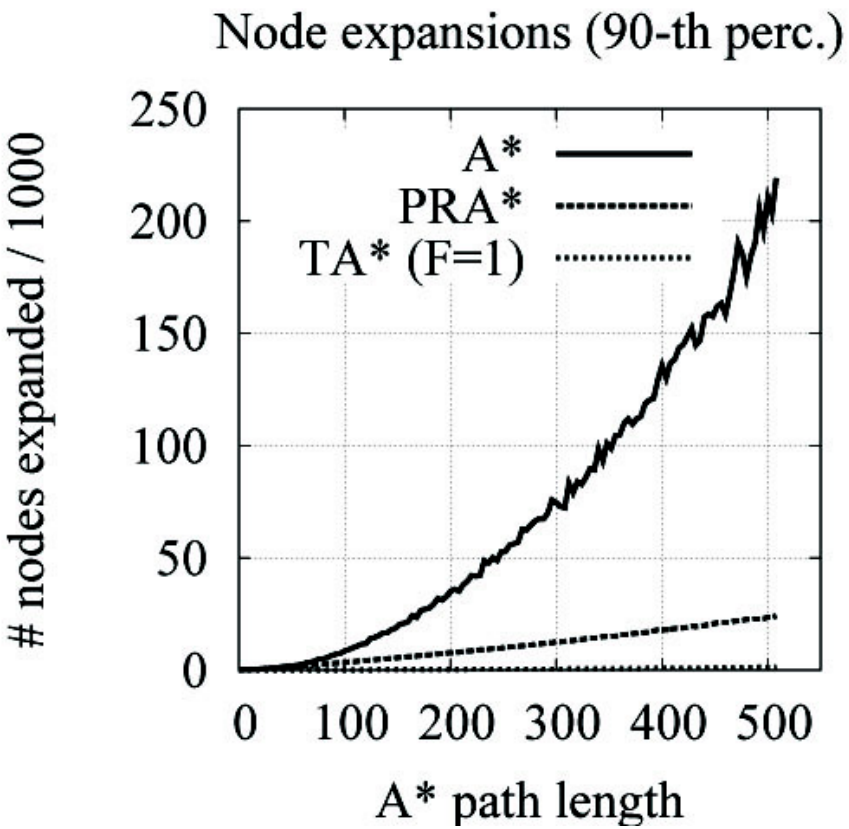
First paths found ($F = 1$) by TA* and TRA* (not searching duplicates)





Speedup and Nodes Expanded

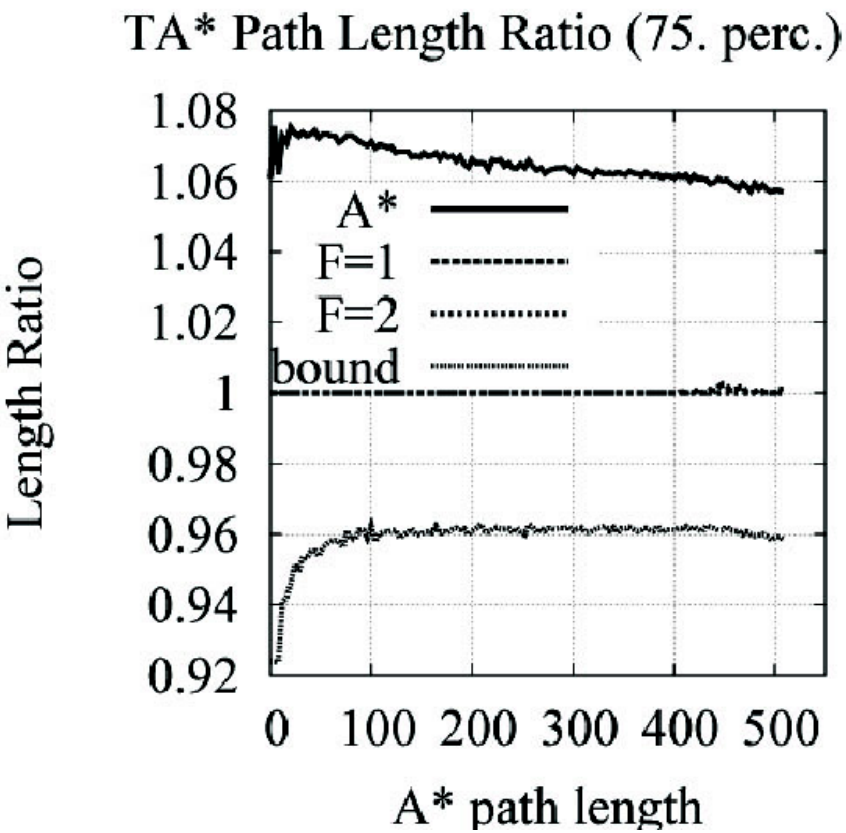


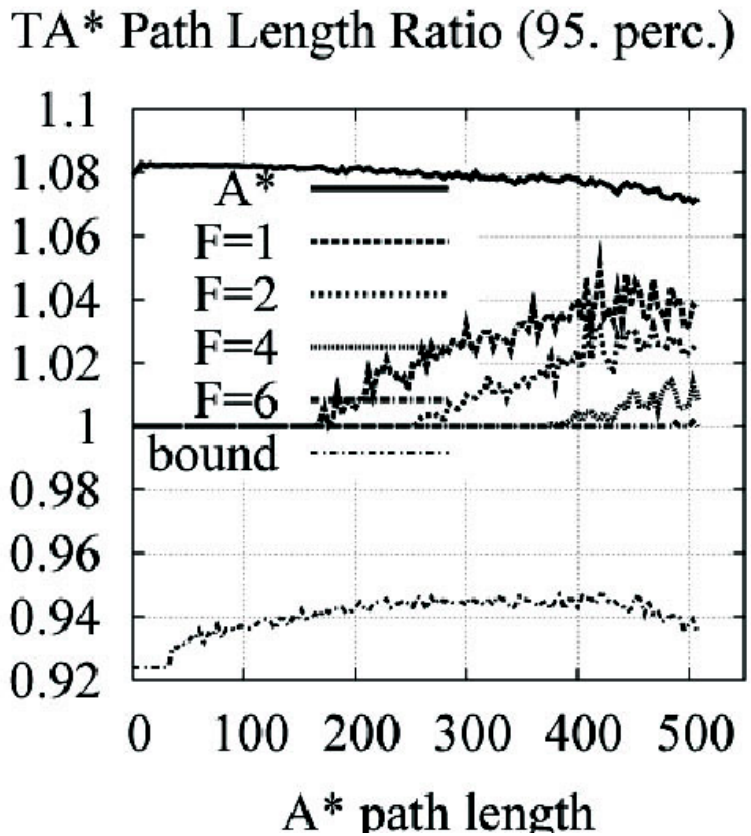


TA* Path Length Ratios Compared To A*

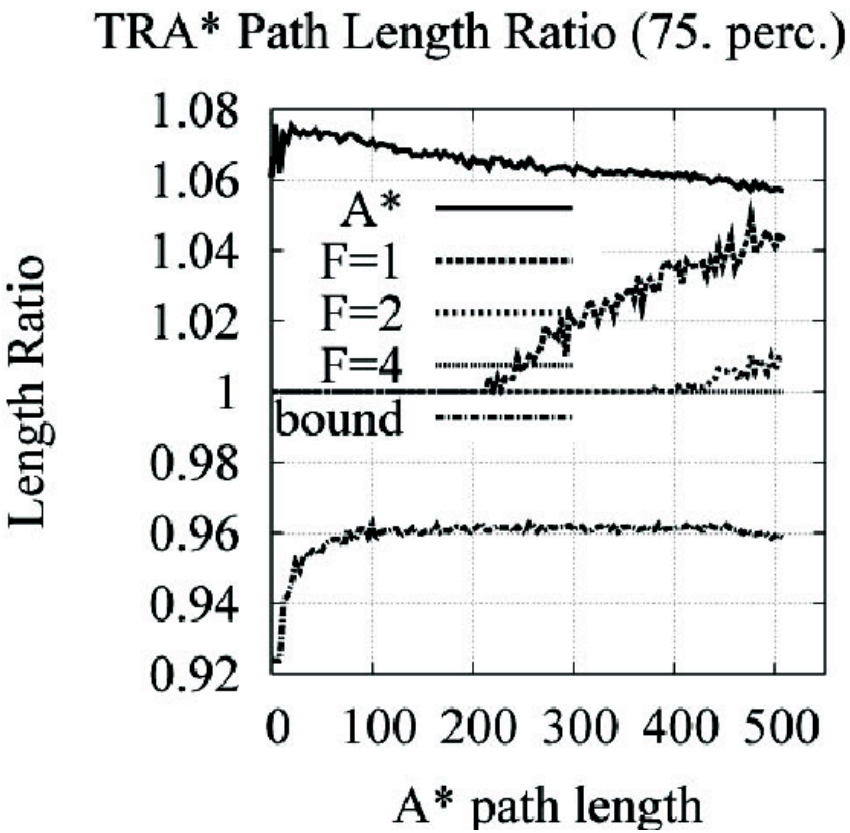
Baseline (ratio 1) is TA* using $F = 40$

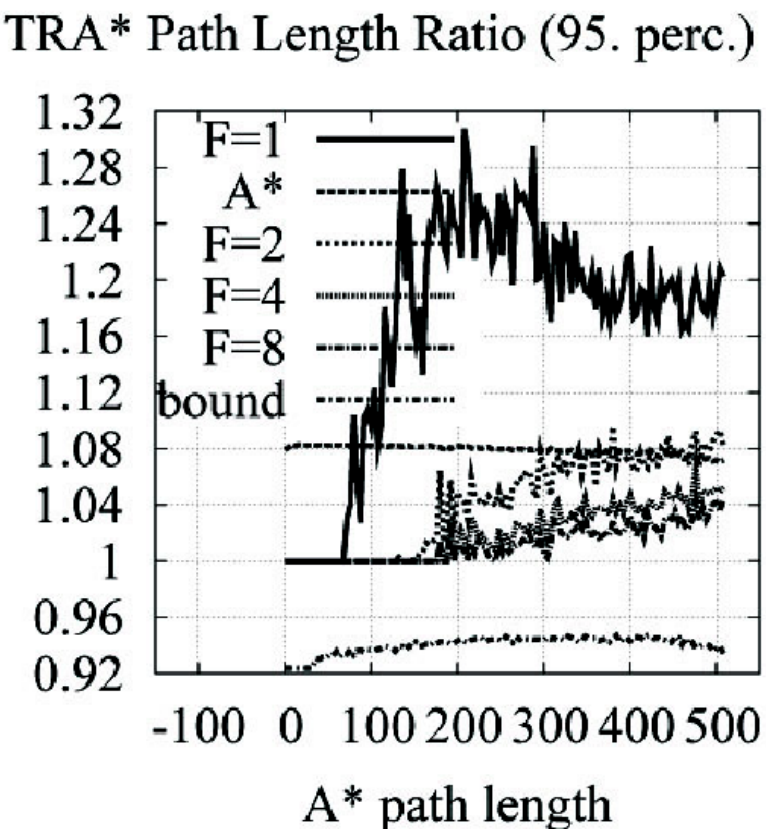
Bound is the ratio lower bound based on the restricted A* paths (one can show: $||d||_{oct}/||d||_2 \leq 1/\cos(22.5°) \approx 1.082$)





TRA* Path Length Ratios Compared To A*





Conclusions

- Triangulations can accurately and efficiently represent polygonal environments
- Triangulations offer unique possibilities for path planning for non-point (especially circular) objects
- Triangulation-based path planning finds paths very quickly and can also find optimal paths given a bit more time
- Our abstraction technique identifies useful structures in the environment: dead-ends, corridors, and decision points
- This abstraction can be used to find paths even more quickly, only depending on the number of obstacles
- TA* is used in StarCraft 2!

Future Work

- Channels resulting from TA* or TRA* are useful in pathfinding involving multiple object sizes because channel widths are known
- Terrain analysis is possible with the abstraction information (e.g., identifying choke points)
- More edge annotations can reduce the need for triangulation updates (e.g., enemy presence in corridors)
- It may be useful to construct waypoint graphs from triangulations that produce close to optimal paths running A* just once

Future Work (continued)

Lots of interesting path planning problems left:

- Group and formation path planning, flocking
- Cooperative path planning (avoiding collisions, planning in space-time!)
- Adversarial settings (pursuit/evasion)

Application areas: video games, robot (team) navigation

References

1. AI Game Programming Wisdom Book Series
2. A. Botea, M. Mueller, J. Schaeffer, Near Optimal Hierarchical Path-Finding. Journal of Game Development, vol 1, pp. 1-22, 2004.
3. M. Renee Jansen, M. Buro: HPA* Enhancements. AIIDE 2007: 84-87. 2006. `skatgame.net/mburo/ps/hpaenh.pdf`
4. N. Sturtevant and M. Buro, Partial Pathfinding Using Map Abstraction and Refinement, AAAI Pittsburgh, pp. 1392-1397, 2005, `skatga net/mburo/ps/path.pdf`
5. M. de Berg et al., Computational Geometry, 3rd edition, Springer Verlag 2008
6. M. Kallmann, H. Bieri, D. Thalmann, Fully Dynamic Constraint Delaunay Triangulations, in Geometric Modeling for Scientific Visualization, Springer Verlag 2003
7. M. Kallmann, Pathplanning in Triangulations, IJCAI 2005

References (continued)

8. D. Demyen, Triangulation-Based Pathfinding, MSc. Thesis, 2006, `skatgame.net/mburo/ps/thesis_demyen_2006.pdf` which is summarized in: D. Demyen and M. Buro, Efficient Triangulation-Based Pathfinding, AAAI 2006
9. K. Chen, Robust Dynamic CDT for Pathfinding, MSc. Thesis, CS Department University of Alberta, 2009 `skatgame.net/mburo/ps/thesis_chen_2009.pdf`
10. L. Devroye and C. Lemaire and J.M. Moreau, Expected Time Analysis for Delaunay Point Location, Computational Geometry Vol. 29, Issue 2, 2004 `https://doi.org/10.1016/j.comgeo.2004.02.002`
11. O. Devillers, S. Pion, M. Teillaud, Walking in a Triangulation, International Journal of Foundations of Computer Science, March 2001, DOI:10.1145/378583.378643
12. H. Edelsbrunner, Geometry and topology for mesh generation, 2001, https://doi.org/10.1017/CBO9780511530067