Robot Path Planning

Overview:

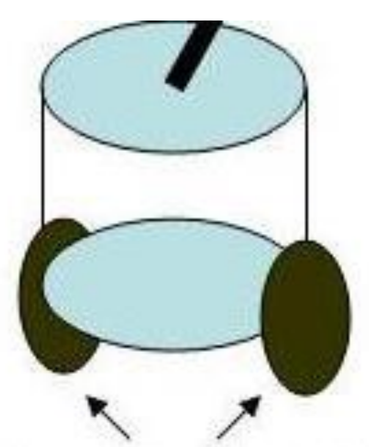
1. Visibility Graphs
2. Voronoi Graphs
3. Potential Fields
4. Sampling-Based Planners
   - PRM: Probabilistic Roadmap Methods
   - RRTs: Rapidly-exploring Random Trees

Robot Path Planning

Things to Consider:

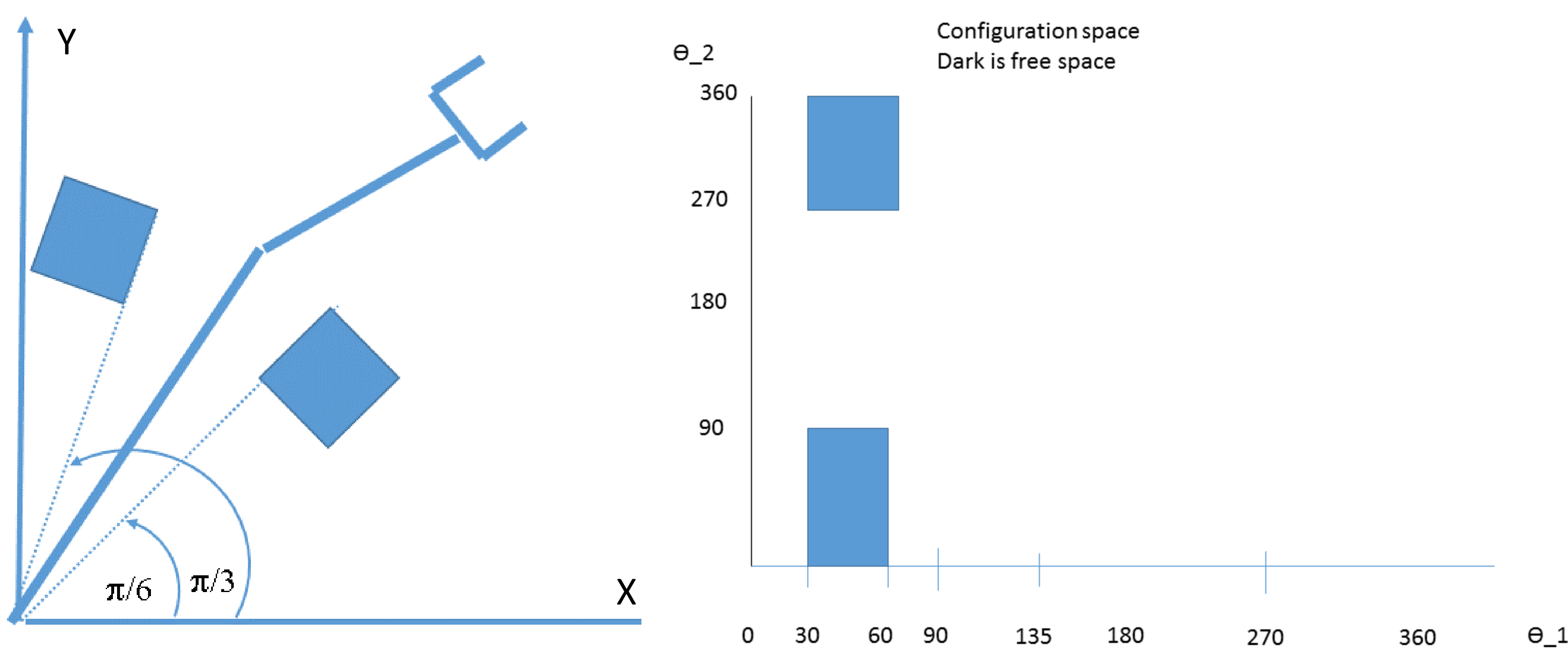
- Spatial reasoning/understanding: robots can have many dimensions in space, obstacles can be complicated
- Global Planning: Do we know the environment *apriori ?*
- Online Local Planning: is environment *dynamic?* Unknown or moving obstacles? Can we compute path “on-the fly”?
- Besides collision-free, should a path be optimal in time, energy or safety?
- Computing exact “safe” paths is provably computationally expensive in 3D – “piano movers” problem
- Kinematic, dynamic, and temporal reasoning may also be required

Configuration Space of a Robot



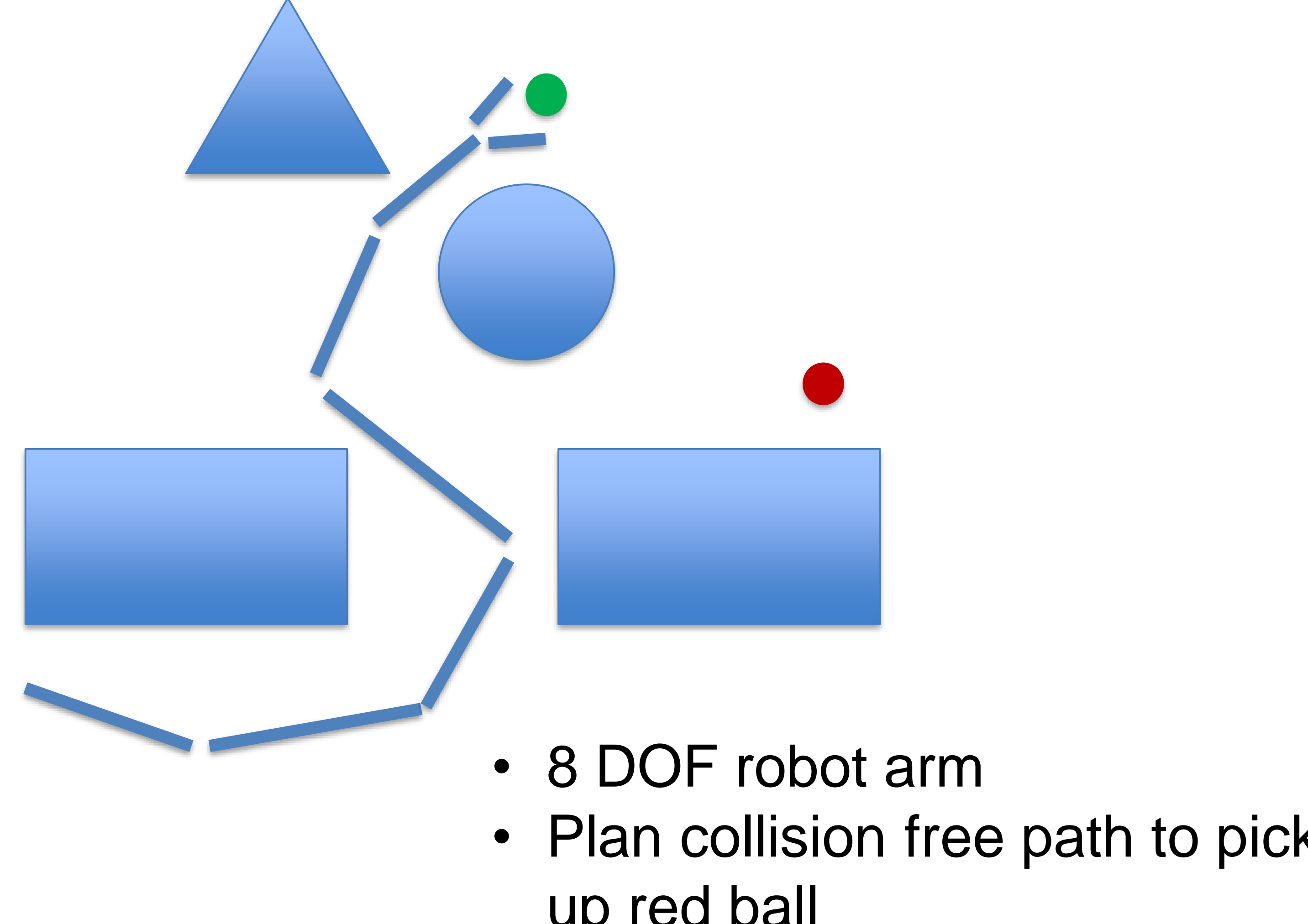
- Configuration Space (C-Space) : Set of parameters that completely describes the robots' state
- Mobile base has 3 Degrees-of-Freedom (DOFs)
- It can translate in the the plane (X,Y) and rotate (Θ)
- C-Space is allowable values of (X,Y,Θ)

Configuration Space: C-Space



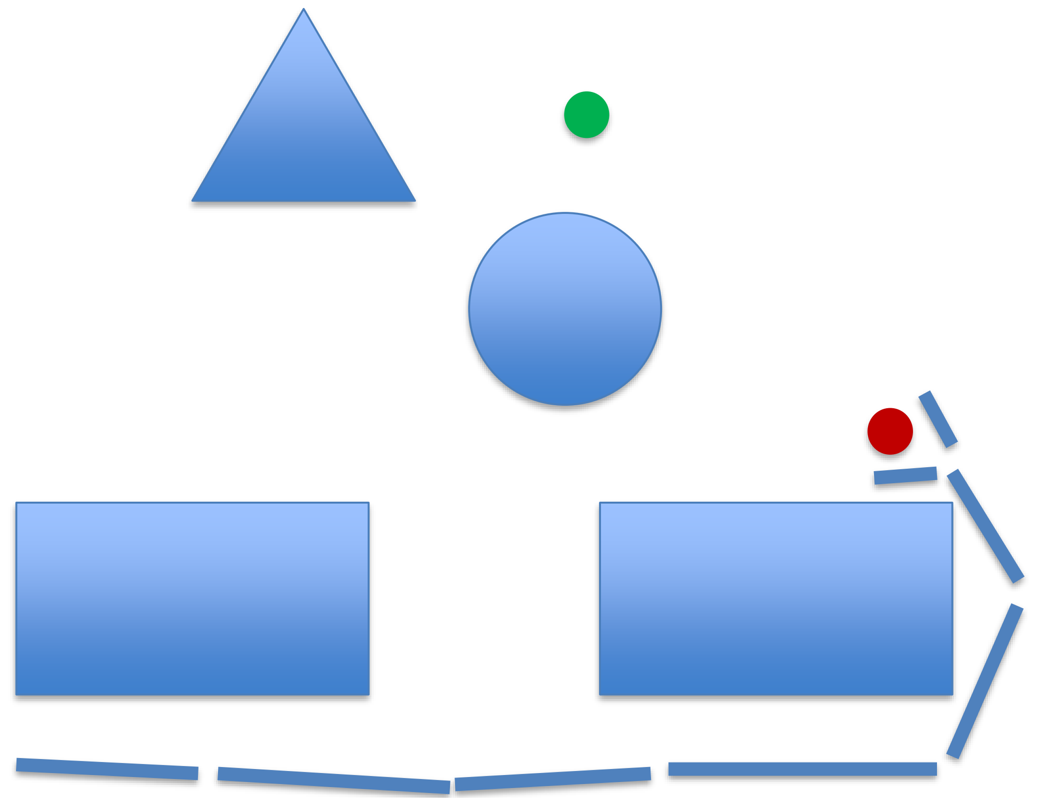
- 2-DOF robot: joints θ_1, θ_2 are the robot's C-Space
- C-Free: values of θ_1, θ_2 where robot is NOT in collision
- C-Free = C-Space – C-Obstacles

Path Planning in Higher Dimensions



- 8 DOF robot arm
- Plan collision free path to pick up red ball

Path Planning in Higher Dimensions



- 8 DOF robot arm
- Plan collision free path to pick up red ball

Path Planning in Higher Dimensions

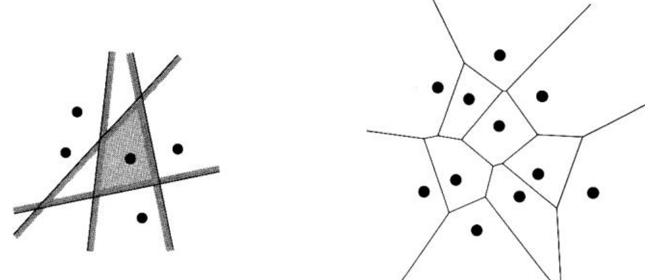
- Humanoid robot has MANY DOFs
- Anthropomorphic Humanoid: Typically >20 joints:
  - 2-6 DOF arms, 2-4 DOF legs, 3 DOF head, 4 DOF torso, plus up to 20 DOF per multi-fingered hand!
- Exact geometric/spatial reasoning difficult
- Complex, cluttered environments also add difficulty

Voronoi Path Planning

- Find paths that are not close to obstacles, but in fact as far away as possible from obstacles.

- This will create a maximal safe path, in that we never come closer to obstacles than we need.

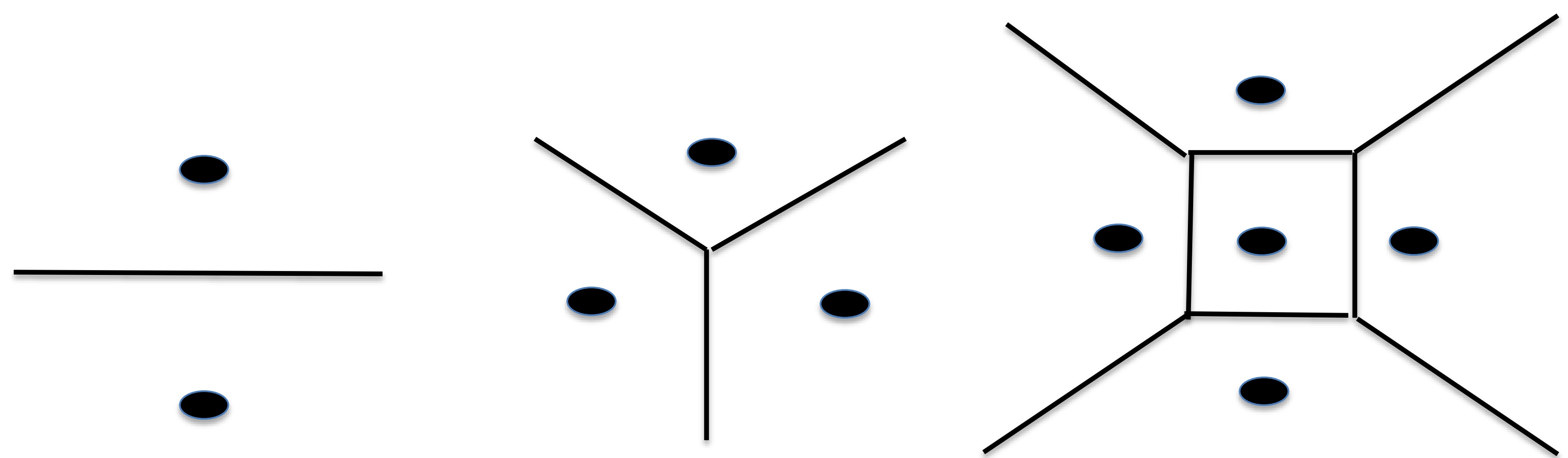
- Voronoi Diagram in the plane. Let $P = \{p_i\}$, set of points in the plane, called *sites.* Voronoi diagram is the sub-division of the plane into N distinct cells, one for each *site*.



- Cell has property that a point q corresponds to a site p_i iff:

  $$dist(q, p_i) < dist(q, p_j) \text{ for all } p_j \in P, j \neq i$$

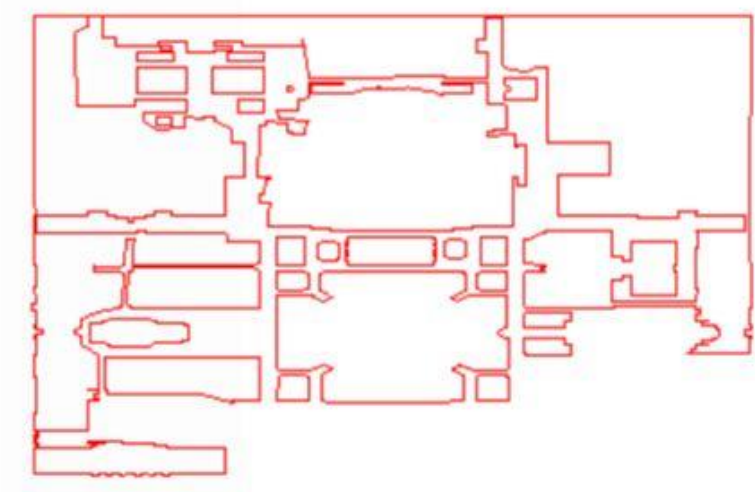
Voronoi Graph



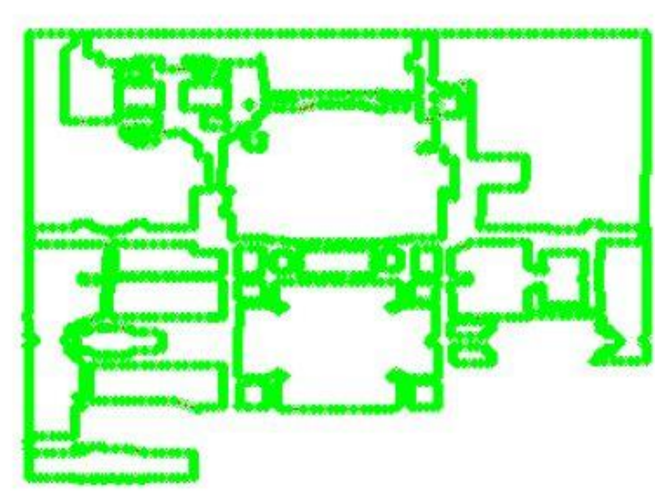
- Intuitively: Edges and vertices are intersections of perpendicular bi-sectors of point-pairs
- Edges are equidistant from 2 points
- Vertices are equidistant from 3 points
- Online demo: http://alexbeutel.com/webgl/voronoi.html

Creating a Voronoi Path

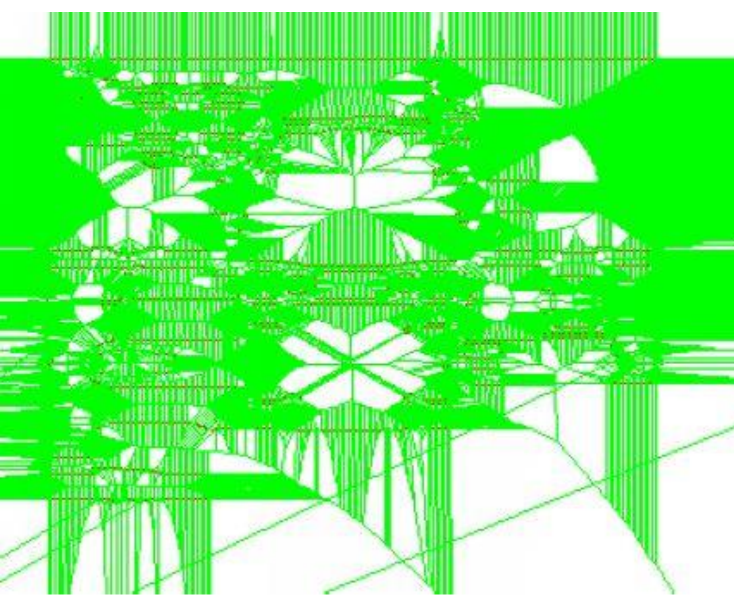
- Approximate the boundaries of the polygonal obstacles as a large number of points
- Compute the Voronoi diagram for this collection of approximating points
- Elminate those Voronoi edges which have one or both endpoints lying inside any of the obstacles
- Remaining Voronoi edges form a good approximation of the generalized Voronoi diagram for the original obstacles in the map
- Locate the robot's starting and stopping points and then compute the Voronoi vertices which are closest to these two points
- Connect start/end points to nearest Voronoi vertex (without collision)
- Use Djikstra or A* graph search to find path vertices.
- Generates a route that for the most part remains equidistant between the obstacles closest to the robot, and gives the robot a relatively safe path along which to travel.



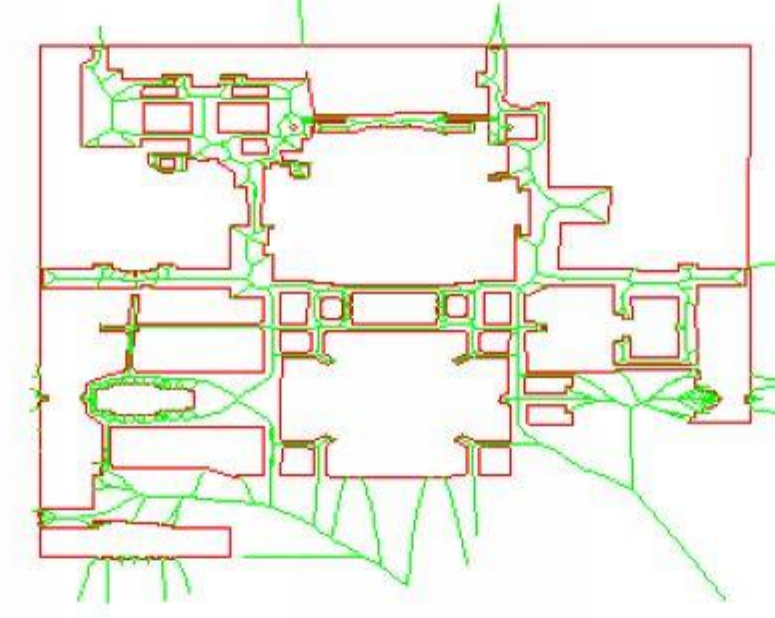
Original Map



Point approximation of boundary

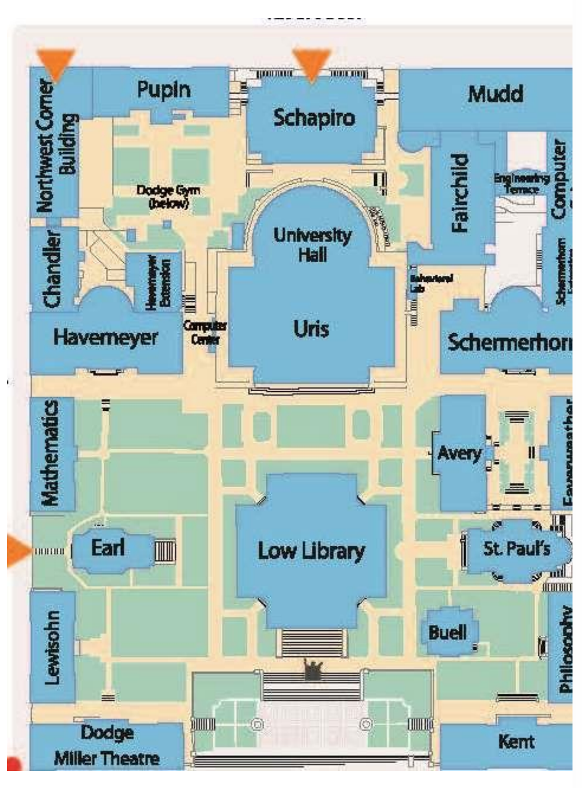


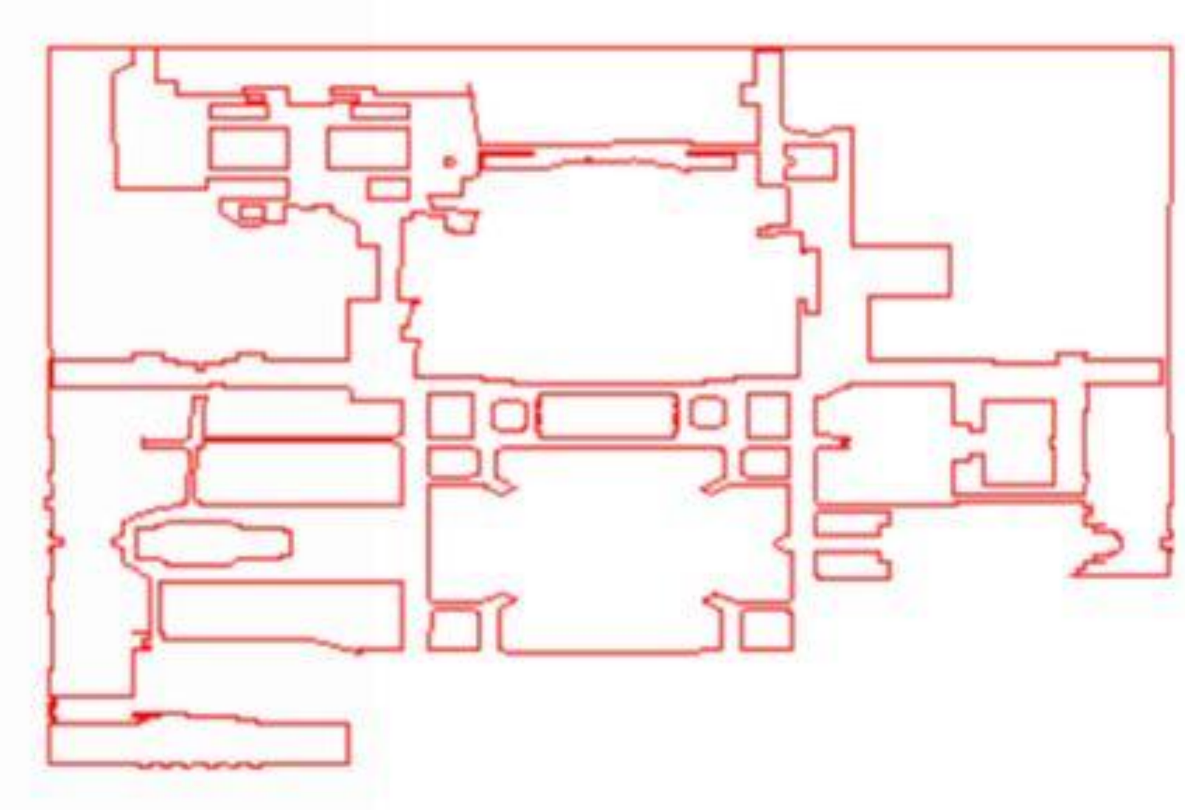
Full Voronoi for Point approximation



Voronoi after eliminating collision edges

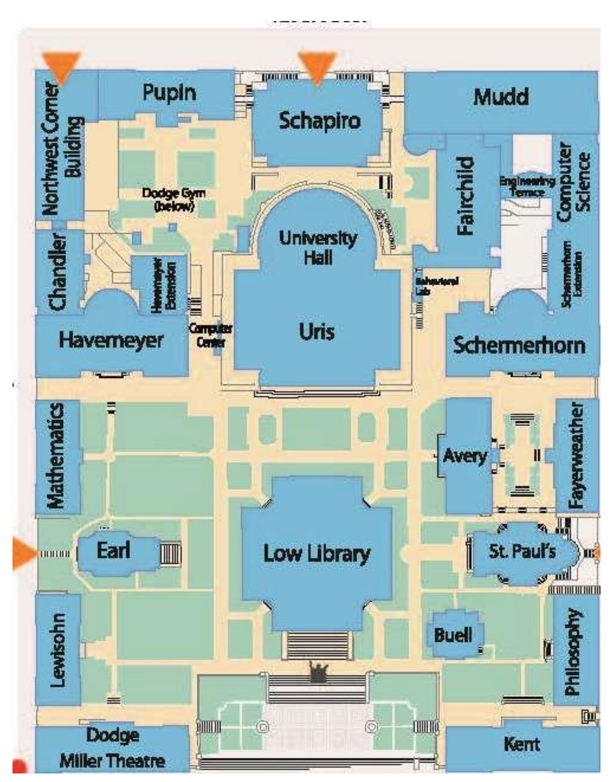
Voronoi Path on Columbia Campus





To find a path:

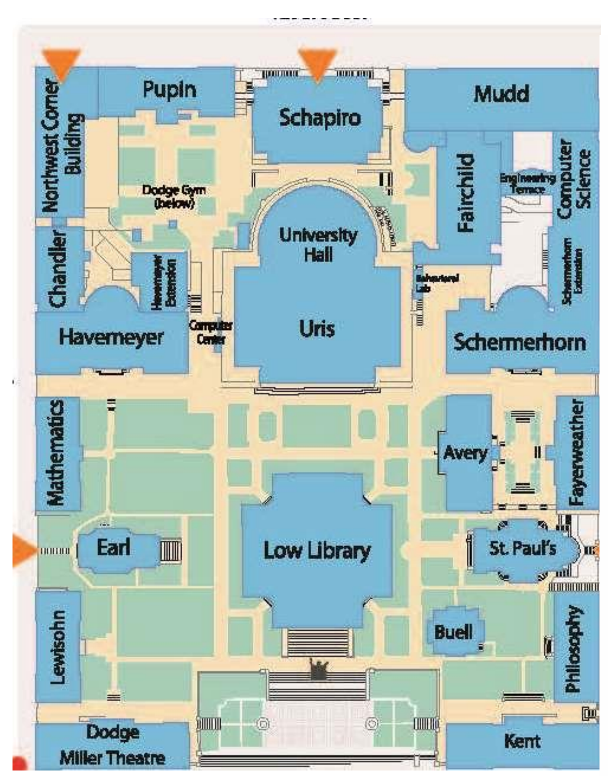
Voronoi Path on Columbia Campus



To find a path:

- Create Voronoi graph - O(N log N) complexity in the plane
- Connect q_start, q goal to graph – local search
- Compute shortest path from q_start to q_goal (A* search)

Voronoi Path on Columbia Campus



start

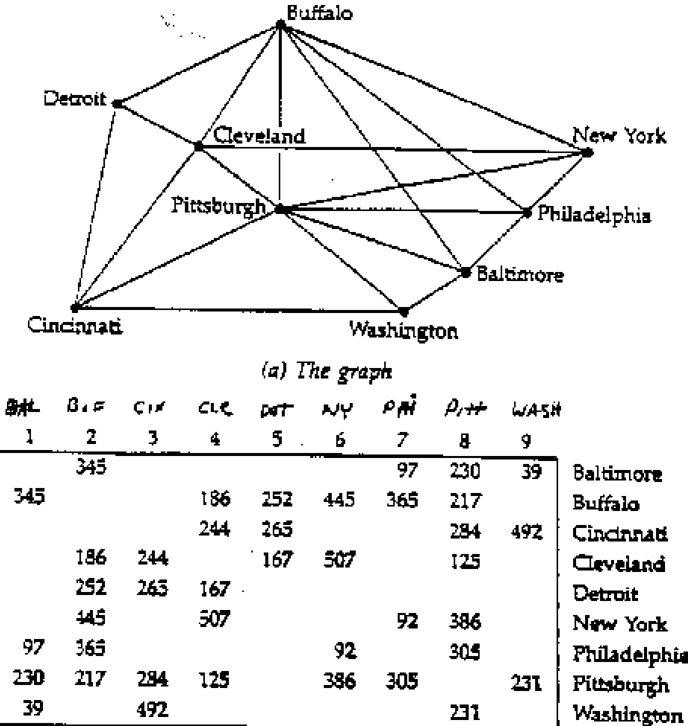
goal

To find a path:

- Create Voronoi graph - O(N log N) complexity in the plane
- Connect q_start, q_goal to graph – local search
- Compute shortest path from q_start to q_goal using A*

Dijkstra’s Algorithm –Shortest Path Graph Search

- We want to compute the shorterst path distance from a source node S to all other nodes. We associate lengths or costs on edges and find the shortest path.
- We can’t use edges with a negative cost. Otherwise, we can take take endless loops to reduce the cost.
- Finding a path from vertex S to vertex T has the same cost as finding a path from vertex S to all other vertices in the graph (within a constant factor).
- If all edge lengths are equal, then the Shortest Path algorithm is equivalent to the breadth-first search algorithm. Breadth first search will expand the nodes of a graph in the minimum cost order from a specified starting vertex (assuming equal edge weights everywhere in the graph).
- **Dijkstra’s Algorithm**: This is a greedy algorithm to find the minimum distance from a node to all other nodes. At each iteration of the algorithm, we choose the minimum distance vertex from all unvisited vertices in the graph,
- There are two kinds of nodes: **Visited** or closed nodes are nodes whose minimum distance from the source node S is known.
- **Unsettled** or open nodes are nodes where we don’t know the minimum distance from S.
- At each iteration we choose the unsettled node V of minimum distance from source S.
- This settles (closes) the node since we know its distance from S. All we have to do now is to update the distance to any unsettled node reachable by an arc from V
- At each iteration we close off another node, and eventually we have all the minimum distances from source node S.



(a) The graph

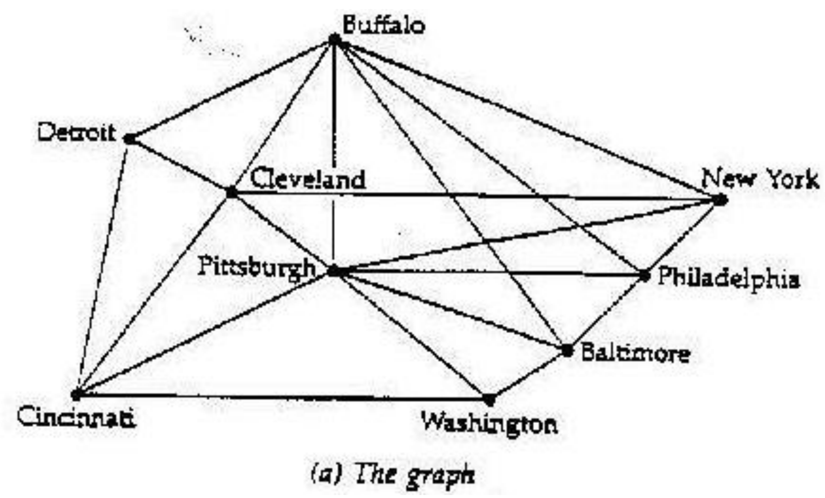
	BAL 1	BUF 2	CIN 3	CLE 4	DET 5	NY 6	PHI 7	PITT 8	WASH 9	
1		345					97	230	39	Baltimore
2	345			186	252	445	365	217		Buffalo
3				244	265			284	492	Cincinnati
4		186	244		167	507		125		Cleveland
5		252	263	167						Detroit
6		445		507			92	386		New York
7	97	365				92		305		Philadelphia
8	230	217	284	125		386	305		231	Pittsburgh
9	39		492					231		Washington

EXAMPLE OF DIJKSTRA'S ALGORITHM

			DISTANCES								
			1	2	3	4	5	6	7	8	9
Iteration	Settled	Selected	Bal	Buff	Cinc	Clev	Det	NYC	Phl	Pitt	Wash
Initial			0	345	inf	inf	inf	inf	97	230	39
1	1	9	0	345	531	inf	inf	inf	97	230	39
2	1,9	7	0	345	531	inf	inf	189	97	230	39
3	1,9,7	6	0	345	531	696	inf	189	97	230	39
4	1,9,7,6	8	0	345	514	355	inf	189	97	230	39
5	1,9,7,6,8,	2	0	345	514	355	597	189	97	230	39
6	1,9,7,6,8,2	4	0	345	514	355	522	189	97	230	39
7	1,9,7,6,8,2,4	3	0	345	514	355	522	189	97	230	39
8	1,9,7,6,8,2,4,3		0	345	514	355	522	189	97	230	39

Figure 5: Example of Dijkstra's algorithm for finding shortest path

	BAL 1	BUF 2	CIN 3	CLE 4	DET 5	NY 6	PHI 7	PITT 8	WASH 9	
1		345					97	230	39	Baltimore
2	345			186	252	445	365	217		Buffalo
3				244	265			284	492	Cincinnati
4		186	244		167	507		125		Cleveland
5		252	265	167						Detroit
6		445		507			92	386		New York
7	97	365				92		305		Philadelphia
8	230	217	284	125		386	305		231	Pittsburgh
9	39		492					231		Washington



(a) The graph

Parent node: At start of search from Baltimore

EXAMPLE OF DIJKSTRA'S ALGORITHM

								DISTANCES				
			1	1	?	?	?	?	1	1	1	
			1	2	3	4	5	6	7	8	9	
Iteration	Settled	Selected	Bal	Buff	Cinc	Clev	Det	NYC	Phl	Pitt	Wash	
Initial			0	345	inf	inf	inf	inf	97	230	39	
1	1	9	0	345	531 ←	inf	inf	inf	97	230 ←	39	
2	1,9	7	0	345	531	inf	inf	189	97	230	39	
3	1,9,7	6	0	345	531	696	inf	189	97	230	39	
4	1,9,7,6	8	0	345	514	355	inf	189	97	230	39	
5	1,9,7,6,8,	2	0	345	514	355	597	189	97	230	39	
6	1,9,7,6,8,2	4	0	345	514	355	522	189	97	230	39	
7	1,9,7,6,8,2,4	3	0	345	514	355	522	189	97	230	39	
8	1,9,7,6,8,2,4,3		0	345	514	355	522	189	97	230	39	

Figure 5: Example of Dijkstra's algorithm for finding shortest path

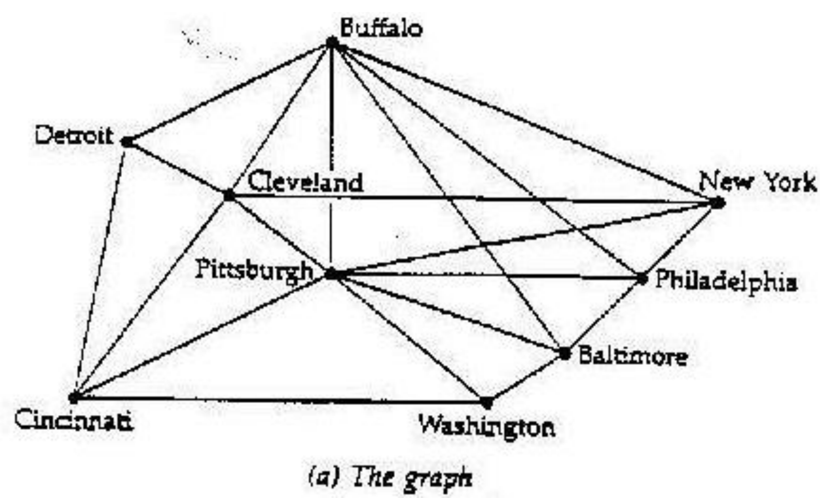
Dijkstra(Graph G, Source_Vertex S)

While Vertices in G remain UNVISITED

Find closest Vertex V that is UNVISITED
Mark V as VISITED
For each UNVISITED vertex W visible from V

If (DIST(S,V) + DIST(V,W)) < DIST(S,W)
then DIST(S,W) = DIST(S,V) + DIST(V,W)

	BAL 1	BUF 2	CIN 3	CLE 4	DET 5	NY 6	PHI 7	PIT 8	WASH 9	
1		345					97	230	39	Baltimore
2	345			186	252	445	365	217		Buffalo
3				244	265			284	492	Cincinnati
4		186	244		167	507		125		Cleveland
5		252	263	167						Detroit
6		445		507			92	386		New York
7	97	365				92		305		Philadelphia
8	230	217	284	125		386	305		231	Pittsburgh
9	39		492					231		Washington



(a) The graph

```
Dijkstra(Graph G, Source_Vertex S)

While Vertices in G remain UNVISITED

    Find closest Vertex V that is UNVISITED
    Mark V as VISITED
    For each UNVISITED vertex W visible from V

        If (DIST(S,V) + DIST(V,W)) < DIST(S,W)
          then DIST(S,W) = DIST(S,V) + DIST(V,W)
```

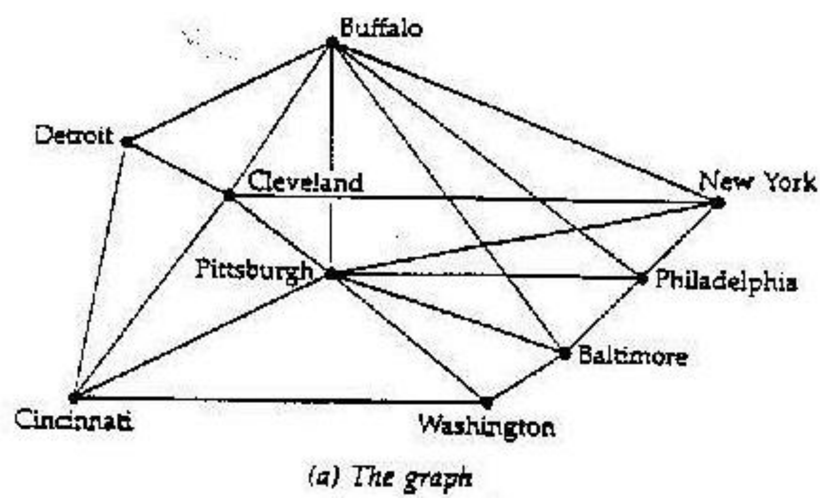
Parent node: after opening up Washington node (#9)

EXAMPLE OF DIJKSTRA'S ALGORITHM

			1	1	9	?	?	?	1	1	1
			DISTANCES								
			1	2	3	4	5	6	7	8	9
Iteration	Settled	Selected	Bal	Buff	Cinc	Clev	Det	NYC	Phl	Pitt	Wash
Initial			0	345	inf	inf	inf	inf	97	230	39
1	1	9	0	345	531 ←	inf	inf	inf	97	230 ←	39
2	1,9	7	0	345	531	inf	inf	189	97	230	39
3	1,9,7	6	0	345	531	696	inf	189	97	230	39
4	1,9,7,6	8	0	345	514	355	inf	189	97	230	39
5	1,9,7,6,8,	2	0	345	514	355	597	189	97	230	39
6	1,9,7,6,8,2	4	0	345	514	355	522	189	97	230	39
7	1,9,7,6,8,2,4	3	0	345	514	355	522	189	97	230	39
8	1,9,7,6,8,2,4,3		0	345	514	355	522	189	97	230	39

Figure 5: Example of Dijkstra's algorithm for finding shortest path

	BAL 1	BUF 2	CIN 3	CLE 4	DET 5	NY 6	PHI 7	PITT 8	WASH 9	
1		345					97	230	39	Baltimore
2	345			186	252	445	365	217		Buffalo
3				244	265			284	492	Cincinnati
4		186	244		167	507		125		Cleveland
5		252	263	167						Detroit
6		445		507			92	386		New York
7	97	365				92		305		Philadelphia
8	230	217	284	125		386	305		231	Pittsburgh
9	39		492					231		Washington



(a) The graph

Dijkstra(Graph G, Source_Vertex S)

While Vertices in G remain UNVISITED

Find closest Vertex V that is UNVISITED
Mark V as VISITED
For each UNVISITED vertex W visible from V

If (DIST(S,V) + DIST(V,W)) < DIST(S,W)
then DIST(S,W) = DIST(S,V) + DIST(V,W)

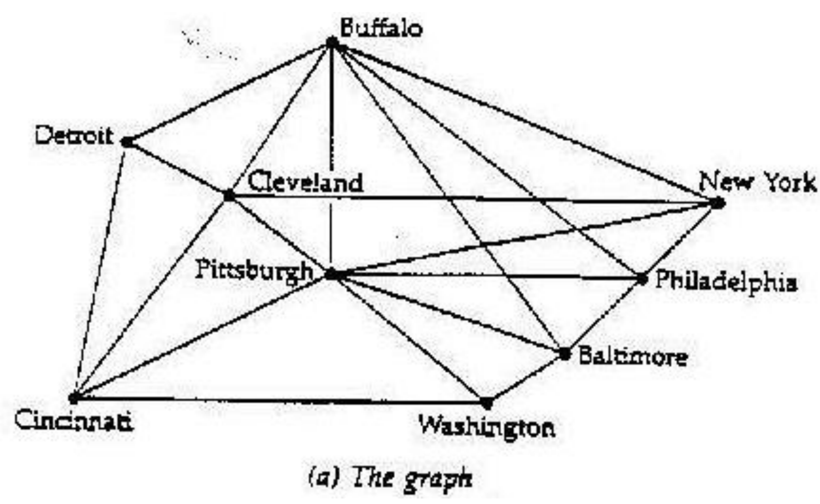
Parent node: after opening up Philadelphia Node (#7)

EXAMPLE OF DIJKSTRA'S ALGORITHM

			1	1	9	?	?	7	1	1	1
			DISTANCES								
			1	2	3	4	5	6	7	8	9
Iteration	Settled	Selected	Bal	Buff	Cinc	Clev	Det	NYC	Phl	Pitt	Wash
Initial			0	345	inf	inf	inf	inf	97	230	39
1	1	9	0	345	531	inf	inf	inf	97	230	39
2	1,9	7	0	345	531	inf	inf	189 ←	97	230	39
3	1,9,7	6	0	345	531	696	inf	189	97	230	39
4	1,9,7,6	8	0	345	514	355	inf	189	97	230	39
5	1,9,7,6,8,	2	0	345	514	355	597	189	97	230	39
6	1,9,7,6,8,2	4	0	345	514	355	522	189	97	230	39
7	1,9,7,6,8,2,4	3	0	345	514	355	522	189	97	230	39
8	1,9,7,6,8,2,4,3		0	345	514	355	522	189	97	230	39

Figure 5: Example of Dijkstra's algorithm for finding shortest path

	BAL 1	BUF 2	CIN 3	CLE 4	DET 5	NY 6	PHI 7	PITT 8	WASH 9	
1		345					97	230	39	Baltimore
2	345			186	252	445	365	217		Buffalo
3				244	265			284	492	Cincinnati
4		186	244		167	507		125		Cleveland
5		252	265	167						Detroit
6		445		507			92	386		New York
7	97	365				92		305		Philadelphia
8	230	217	284	125		386	305		231	Pittsburgh
9	39		492					231		Washington



(a) The graph

Dijkstra(Graph G, Source_Vertex S)

While Vertices in G remain UNVISITED

Find closest Vertex V that is UNVISITED
Mark V as VISITED
For each UNVISITED vertex W visible from V

If (DIST(S,V) + DIST(V,W)) < DIST(S,W)
then DIST(S,W) = DIST(S,V) + DIST(V,W)

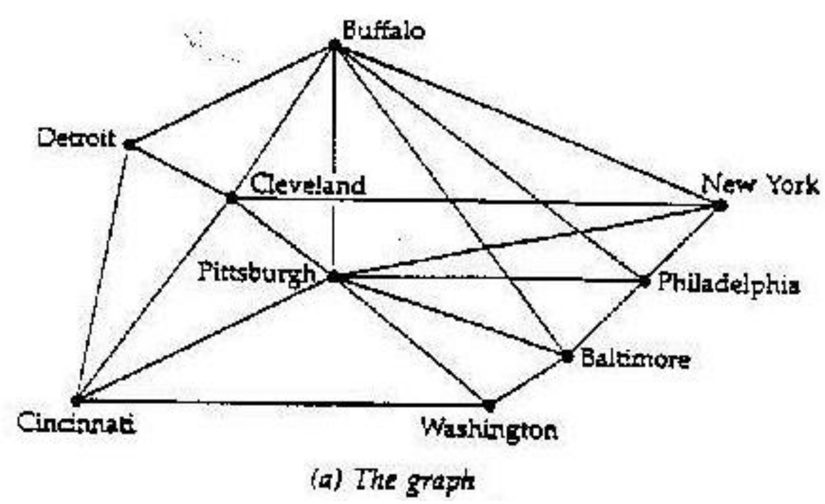
Parent node: after opening up NY Node (#6)

EXAMPLE OF DIJKSTRA'S ALGORITHM

			1	1	9	6	?	7	1	1	1
			DISTANCES								
			1	2	3	4	5	6	7	8	9
Iteration	Settled	Selected	Bal	Buff	Cinc	Clev	Det	NYC	Phl	Pitt	Wash
Initial			0	345	inf	inf	inf	inf	97	230	39
1	1	9	0	345	531	inf	inf	inf	97	230	39
2	1,9	7	0	345	531	inf	inf	189	97	230	39
3	1,9,7	6	0	345	531	696 ←	inf	189	97	230	39
4	1,9,7,6	8	0	345	514	355	inf	189	97	230	39
5	1,9,7,6,8,	2	0	345	514	355	597	189	97	230	39
6	1,9,7,6,8,2	4	0	345	514	355	522	189	97	230	39
7	1,9,7,6,8,2,4	3	0	345	514	355	522	189	97	230	39
8	1,9,7,6,8,2,4,3		0	345	514	355	522	189	97	230	39

Figure 5: Example of Dijkstra's algorithm for finding shortest path

	BAL 1	BUF 2	CIN 3	CLE 4	DET 5	NY 6	PHI 7	PITT 8	WASH 9	
1		345					97	230	39	Baltimore
2	345			186	252	445	365	217		Buffalo
3				244	265			284	492	Cincinnati
4		186	244		167	507		125		Cleveland
5		252	263	167						Detroit
6		445		507			92	386		New York
7	97	365				92		305		Philadelphia
8	230	217	284	125		386	305		231	Pittsburgh
9	39		492					231		Washington



(a) The graph

Dijkstra(Graph G, Source_Vertex S)

While Vertices in G remain UNVISITED

Find closest Vertex V that is UNVISITED
Mark V as VISITED
For each UNVISITED vertex W visible from V

If (DIST(S,V) + DIST(V,W)) < DIST(S,W)
then DIST(S,W) = DIST(S,V) + DIST(V,W)

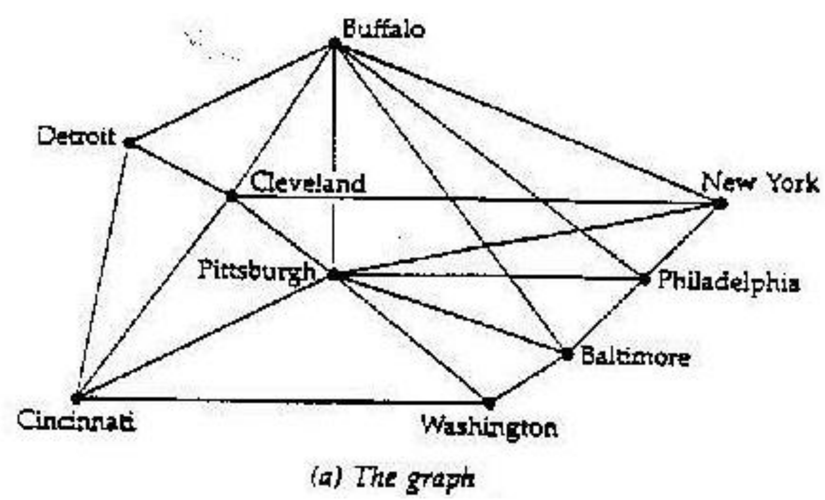
Parent node: after opening up PITT Node (#8)

EXAMPLE OF DIJKSTRA'S ALGORITHM

			1	1	8	8	?	7	1	1	1
			DISTANCES								
			1	2	3	4	5	6	7	8	9
Iteration	Settled	Selected	Bal	Buff	Cinc	Clev	Det	NYC	Phi	Pitt	Wash
Initial			0	345	inf	inf	inf	inf	97	230	39
1	1	9	0	345	531	inf	inf	inf	97	230	39
2	1,9	7	0	345	531	inf	inf	189	97	230	39
3	1,9,7	6	0	345	531	696	inf	189	97	230	39
4	1,9,7,6	8	0	345	514 ←	355 ←	inf	189	97	230	39
5	1,9,7,6,8,	2	0	345	514	355	597	189	97	230	39
6	1,9,7,6,8,2	4	0	345	514	355	522	189	97	230	39
7	1,9,7,6,8,2,4	3	0	345	514	355	522	189	97	230	39
8	1,9,7,6,8,2,4,3		0	345	514	355	522	189	97	230	39

Figure 5: Example of Dijkstra's algorithm for finding shortest path

	BAL 1	BUF 2	CIN 3	CLE 4	DET 5	NY 6	PHI 7	PITT 8	WASH 9	
1		345					97	230	39	Baltimore
2	345			186	252	445	365	217		Buffalo
3				244	265			284	492	Cincinnati
4		186	244		167	507		125		Cleveland
5		252	265	167						Detroit
6		445		507			92	386		New York
7	97	365				92		305		Philadelphia
8	230	217	284	125		386	305		231	Pittsburgh
9	39		492					231		Washington



(a) The graph

Dijkstra(Graph G, Source_Vertex S)

While Vertices in G remain UNVISITED

Find closest Vertex V that is UNVISITED
Mark V as VISITED
For each UNVISITED vertex W visible from V

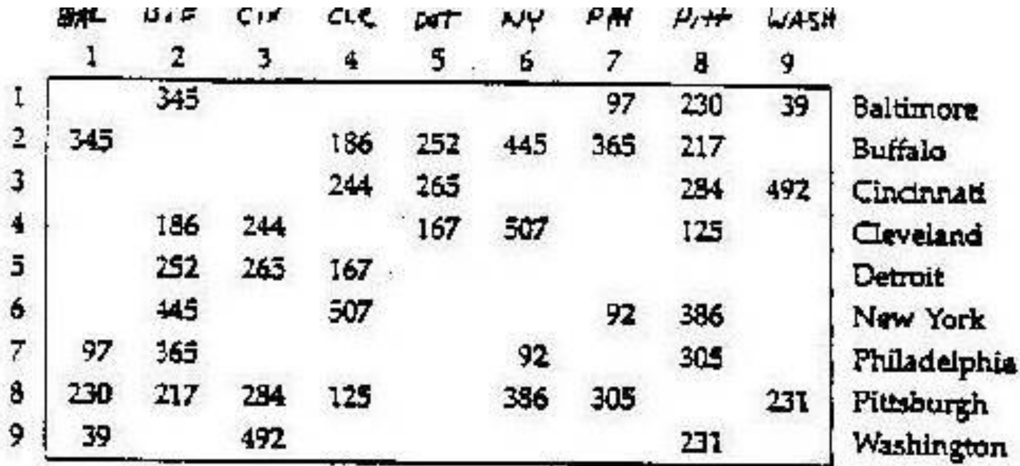
If (DIST(S,V) + DIST(V,W)) < DIST(S,W)
then DIST(S,W) = DIST(S,V) + DIST(V,W)

Parent node: after opening up Buff Node (#2)

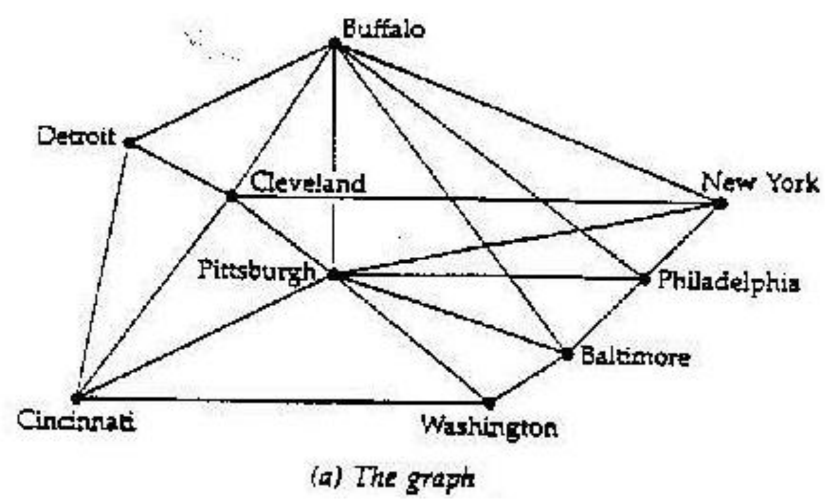
EXAMPLE OF DIJKSTRA'S ALGORITHM

			1	1	8	8	2	7	1	1	1
			DISTANCES								
			1	2	3	4	5	6	7	8	9
Iteration	Settled	Selected	Bal	Buff	Cinc	Clev	Det	NYC	Phl	Pitt	Wash
Initial			0	345	inf	inf	inf	inf	97	230	39
1	1	9	0	345	531	inf	inf	inf	97	230	39
2	1,9	7	0	345	531	inf	inf	189	97	230	39
3	1,9,7	6	0	345	531	696	inf	189	97	230	39
4	1,9,7,6	8	0	345	514	355	inf	189	97	230	39
5	1,9,7,6,8,	2	0	345	514	355	597 ←	189	97	230	39
6	1,9,7,6,8,2	4	0	345	514	355	522	189	97	230	39
7	1,9,7,6,8,2,4	3	0	345	514	355	522	189	97	230	39
8	1,9,7,6,8,2,4,3		0	345	514	355	522	189	97	230	39

Figure 5: Example of Dijkstra's algorithm for finding shortest path



	BAL 1	BUF 2	CIN 3	CLE 4	DET 5	NY 6	PHI 7	PITT 8	WASH 9	
1		345					97	230	39	Baltimore
2	345			186	252	445	365	217		Buffalo
3				244	265			284	492	Cincinnati
4		186	244		167	507		125		Cleveland
5		252	265	167						Detroit
6		445		507			92	386		New York
7	97	365				92		305		Philadelphia
8	230	217	284	125		386	305		231	Pittsburgh
9	39		492					231		Washington



(a) The graph

Once all nodes are settled, follow parent links to find path:
e.g. Detroit – Cleveland - Pittsburgh - Baltimore

Parent node: after opening up CLE Node (#4)

EXAMPLE OF DIJKSTRA'S ALGORITHM

			1	1	8	8	4	7	1	1	1
						DISTANCES					
			1	2	3	4	5	6	7	8	9
Iteration	Settled	Selected	Bal	Buff	Cinc	Clev	Det	NYC	Phl	Pitt	Wash
Initial			0	345	inf	inf	inf	inf	97	230	39
1	1	9	0	345	531	inf	inf	inf	97	230	39
2	1,9	7	0	345	531	inf	inf	189	97	230	39
3	1,9,7	6	0	345	531	696	inf	189	97	230	39
4	1,9,7,6	8	0	345	514	355	inf	189	97	230	39
5	1,9,7,6,8,	2	0	345	514	355	597	189	97	230	39
6	1,9,7,6,8,2	4	0	345	514	355	522 ←	189	97	230	39
7	1,9,7,6,8,2,4	3	0	345	514	355	522	189	97	230	39
8	1,9,7,6,8,2,4,3		0	345	514	355	522	189	97	230	39

Figure 5: Example of Dijkstra's algorithm for finding shortest path

Dijkstra(Graph G, Source_Vertex S)

While Vertices in G remain UNVISITED

Find closest Vertex V that is UNVISITED
Mark V as VISITED
For each UNVISITED vertex W visible from V

If (DIST(S,V) + DIST(V,W)) < DIST(S,W)
then DIST(S,W) = DIST(S,V) + DIST(V,W)

6. Pseudo Code for Dijkstra's Algorithm (see figure 5)

```
Note: initialize all distances from Start vertex S
to each visible vertex. All unknown distances assumed
infinite.  Mark Start Vertex S as VISITED, DIST=0

Dijkstra(Graph G, Source_Vertex S)
{
  While Vertices in G remain UNVISITED
  {
        Find closest Vertex V that is UNVISITED
        Mark V as VISITED
        For each UNVISITED vertex W visible from V
        {
            If (DIST(S,V) + DIST(V,W)) < DIST(S,W)
               then DIST(S,W) =  DIST(S,V) + DIST(V,W)
        }
  }
}
```

7. Sketch of Proof that Dijkstra's Algorithm Produces Min Cost Path

   (a) At each stage of the algorithm, we settle a new node V and that will be the minimum distance from the source node S to V. To prove this, assume the algorithm *does not* report the minimum distance to a node, and let V be the first such node reported as settled yet whose distance reported by Dijkstra, $Dist(V)$, is not a minimum.

   (b) If $Dist(V)$ is not the minimum cost, then there must be an unsettled node X such that $Dist(X)+Edge(X,V) < Dist(V)$. However, this implies that $Dist(X) < Dist(V)$, and if this were so, Dijkstra's algorithm would have chosen to settle node X before we settled node V since it has a smaller distance value from S. Therefore, $Dist(X)$ cannot be $< Dist(V)$, and $Dist(V)$ is the minium cost path from S to V.

8. Improving Dijkstra: A* Algorithm – Heuristic Search

The A* algorithm searches a graph efficiently, with respect to a chosen heuristic. If the heuristic is "good," then the search is efficient; if the heuristic is "bad," although a path will be found, its search will take more time than probably required and possibly return a suboptimal path. The path cost at a node is F=G+H, where G is the minimum distance to the current node from the start node, and H is the heuristic cost of traveling from the current node to the goal. A* will produce an optimal path if it's heuristic is optimistic. An optimistic, or admissible, heuristic always returns a value less than or equal to the actual cost of the shortest path from the current node to the goal node within the graph.

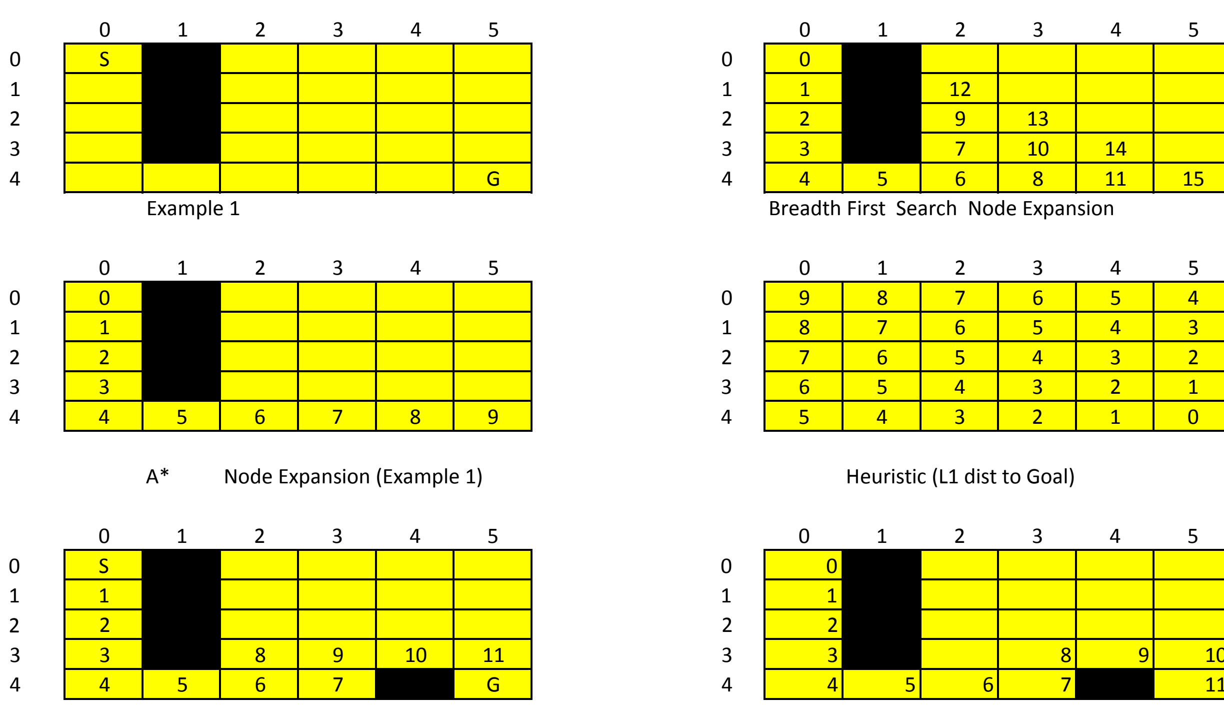
The A* search has a priority queue which contains a list of nodes sorted by priority. This priority is determined by the sum of the distance from the start node to the current node and the heuristic at the current node. The first node to be put into the priority queue is naturally the start node. Next, we expand the start node by popping the start node and putting all adjacent nodes to the start node into the priority queue sorted by their corresponding priorities (path costs). Note that only unvisited nodes are added to the priority queue. At each step, the highest priority node (i.e. least cost node) is dequeued and expanded until the goal is reached. A* is greedy in that it tries to skew the search towards the goal. Breadth first search can be thought of as search with heuristic function H=0 (i.e. no heuristic).

A* Search on 4- neighbor Grid

i **Breadth First search expands more nodes than A***

ii **A* with a heuristic function =0 becomes Breadth First Search**

iii **A* is admissible if heuristic cost is an UNDERESTIMATE of the true cost**



Example 1

Breadth First Search Node Expansion

A* Node Expansion (Example 1)

Heuristic (L1 dist to Goal)

A* Node Expansion (Example 2)

A* Final Path
(follow goal node back via
Opener node to compute path)

OPEN LIST - Example 2

Opener	Node	f	g	h	f	expand order
	[0,0]	0+9	0	9	9	0
0,0	[1,0]	1+8	1	8	9	1
1,0	[2,0]	2+7	2	7	9	2
2,0	[3,0]	3+6	3	6	9	3
3,0	[4,0]	4+5	4	5	9	4
4,0	[4,1]	5+4	5	4	9	5
4,1	[4,2]	6+3	6	3	9	6
4,2	[4,3]	7+2	7	2	9	7
4,2	[3,2]	7+4	7	4	11	8
4,3	[3,3]	8+3	8	3	11	9
3,2	[2,2]	8+5	8	5	13	
3,3	[2,3]	8+4	8	4	12	
3,3	[3,4]	8+2	8	2	10	10
3,4	[3,5]	9+1	9	1	10	11
3,4	[2,4]	9+3	9	3	12	
3,5	[4,5]	goal				

Path Cost= f = g + h
g= min distance traveled to this node
h= heuristic cost to goal from this node
(we are using L1 metric cost on 4-neighbor grid)

A* is admissible if heuristic cost is an UNDERESTIMATE of the true cost: h <= C(i,j)

TopBot: Topological Mobile Robot Localization Using Fast Vision Techniques

Paul Blaer and Peter Allen
Dept. of Computer Science, Columbia University
{psb15, allen}@*cs.columbia.edu*

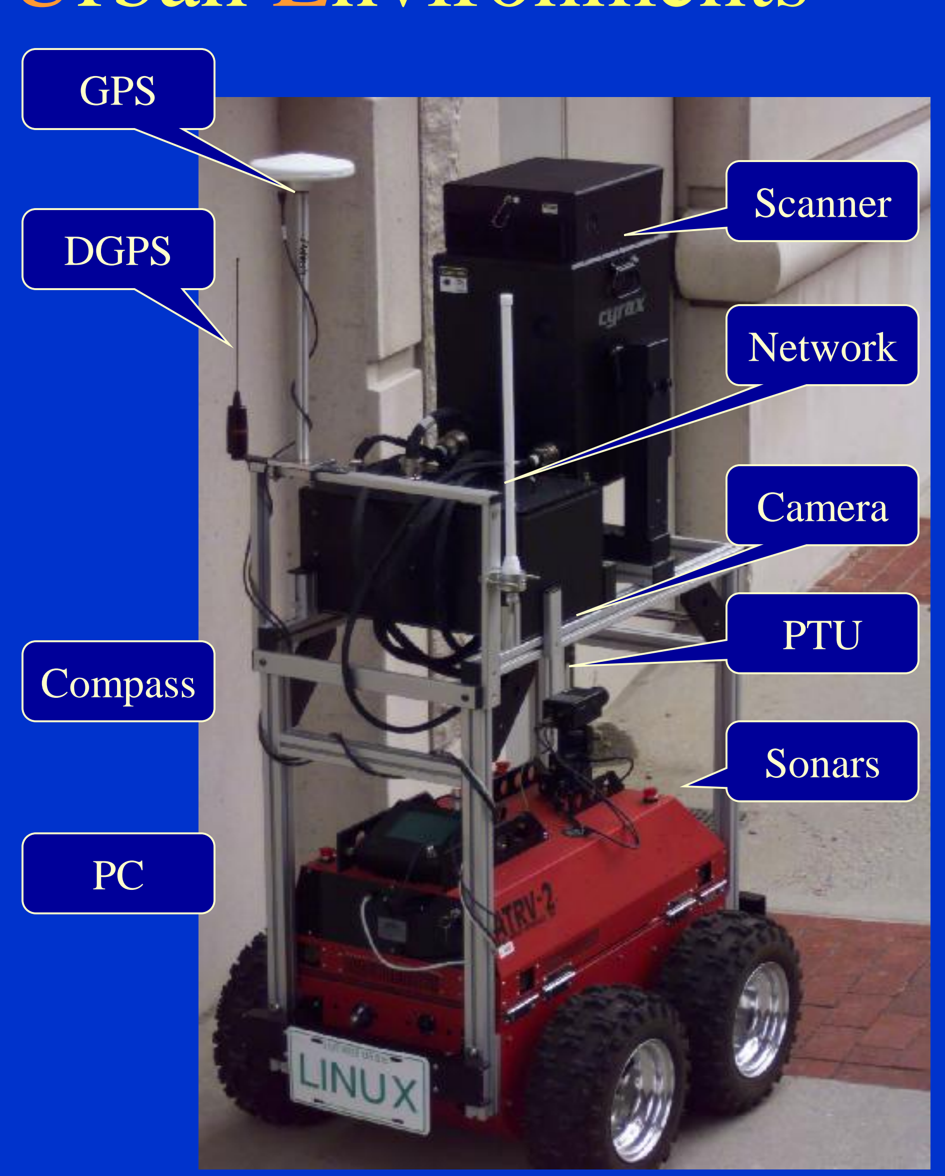
Autonomous Vehicle for Exploration and Navigation in Urban Environments

The AVENUE Robot:

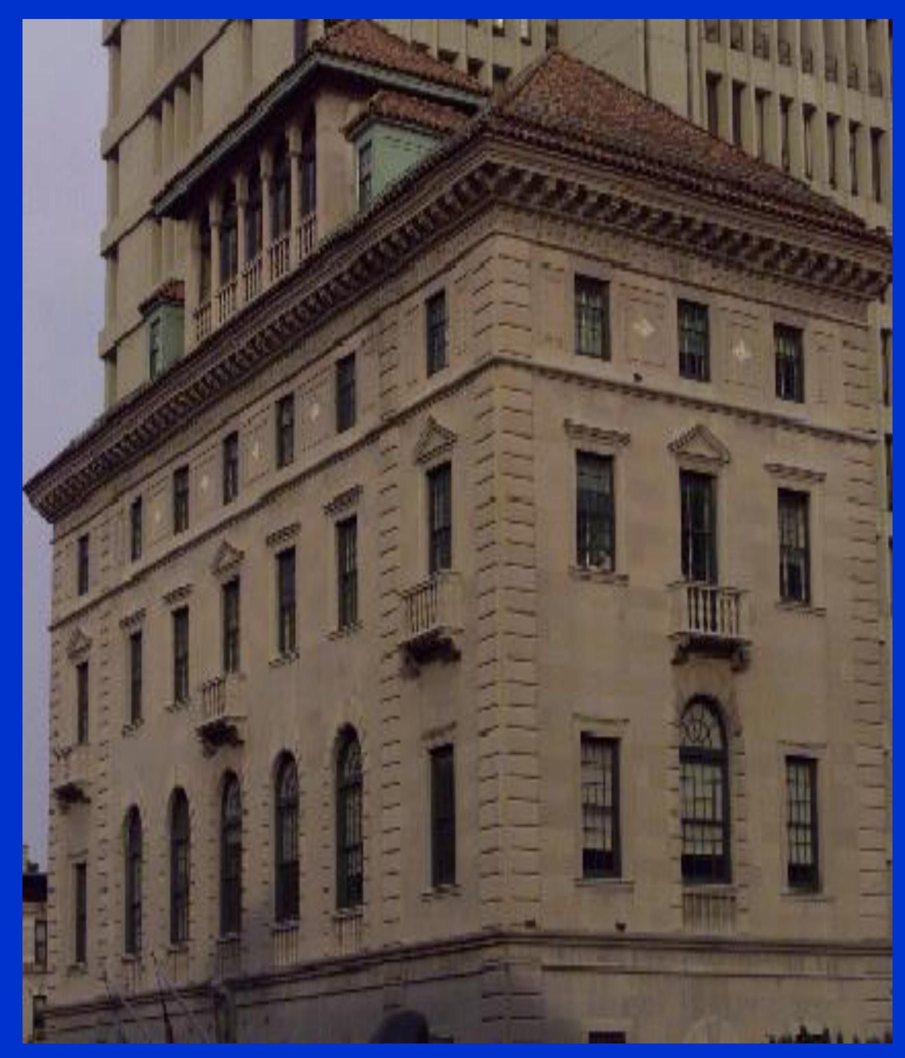
- Autonomous.
- Operates in outdoor urban environments.
- Builds accurate 3-D models.

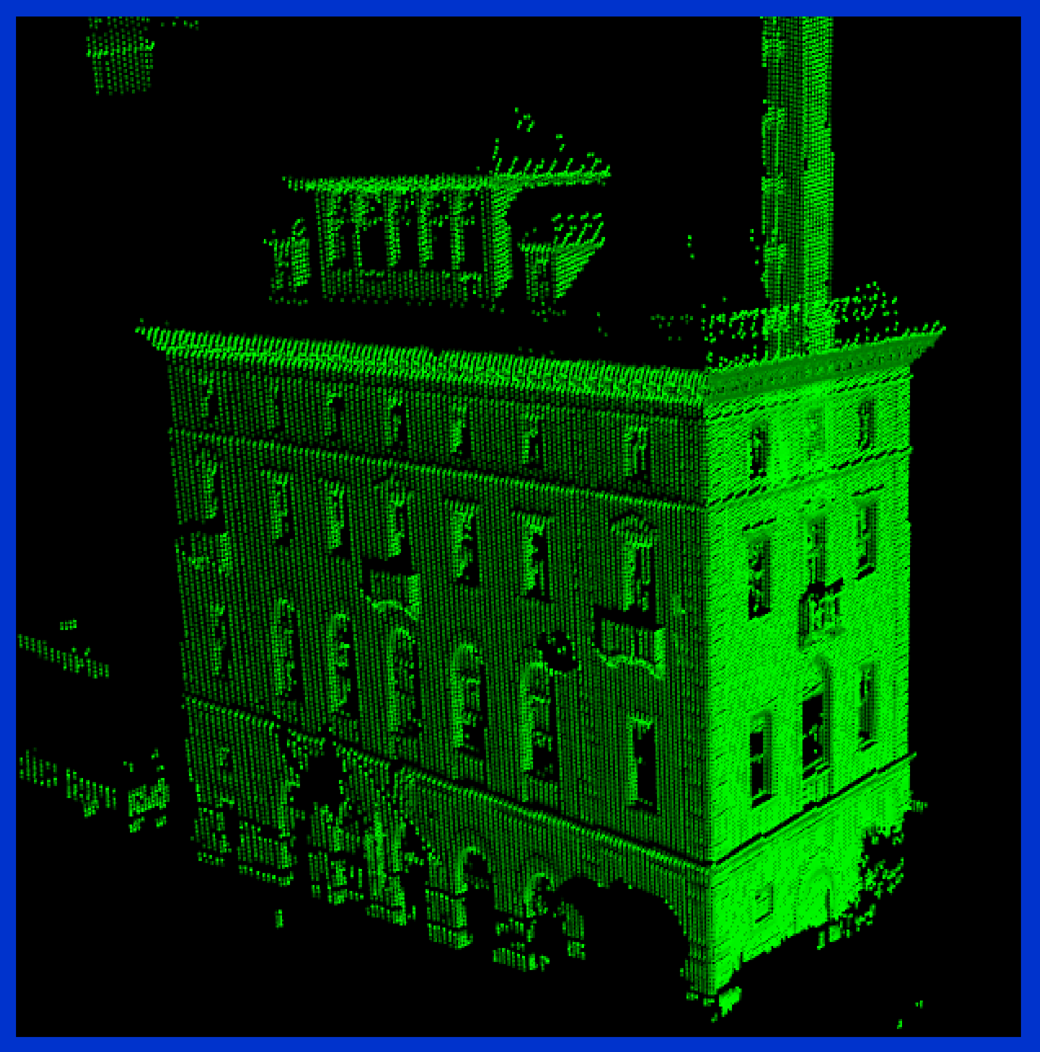
Current Localization Methods:

- Odometry.
- Differential GPS.
- Vision.



Range Scanning Outdoor Structures





Italian House: Textured 3-D Model

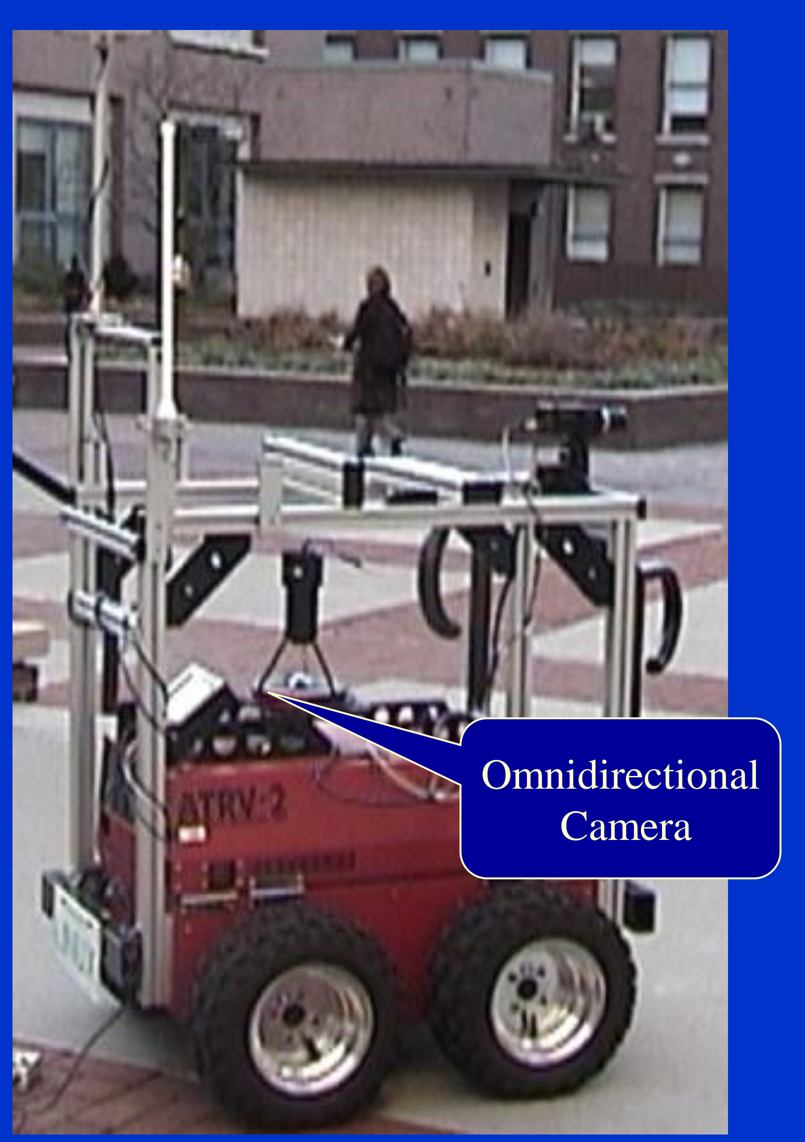
Main Vision System

Georgiev and Allen '02

Topological Localization

- Odometry and GPS can fail.
- Fine vision techniques need an estimate of the robot's current position.

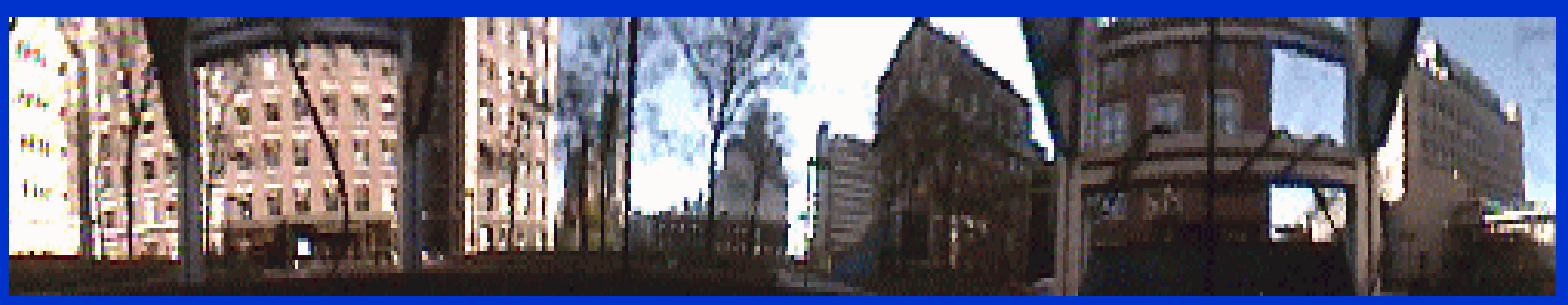
Histogram Matching of Omnidirectional Images:
- Fast.
- Rotationally-invariant.
- Finds the region of the robot.
- This region serves as a starting estimate for the main vision system.



Building the Database

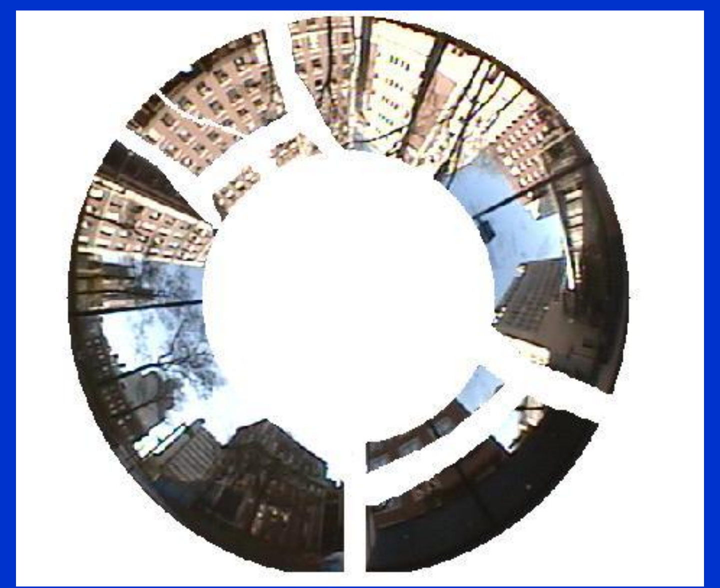
- Divide the environment into a logical set of regions.
- Reference images must be taken in all of these regions.
- The images should be taken in a zig-zag pattern to cover as many different locations as possible.
- Each image is reduced to three 256-bucket histograms, for the red, green, and blue color bands.

An Image From the Omnidirectional Camera



Masking the Image

- The camera points up to get a clear picture of the buildings.
- The camera pointing down would give images of the ground's brick pattern - not useful for histogramming.
- With the camera pointing up, the sun and sky enter the picture and cause major color variations.
- We mask out the center of the image to block out most of the sky.
- We also mask out the superstructure associated with the camera.



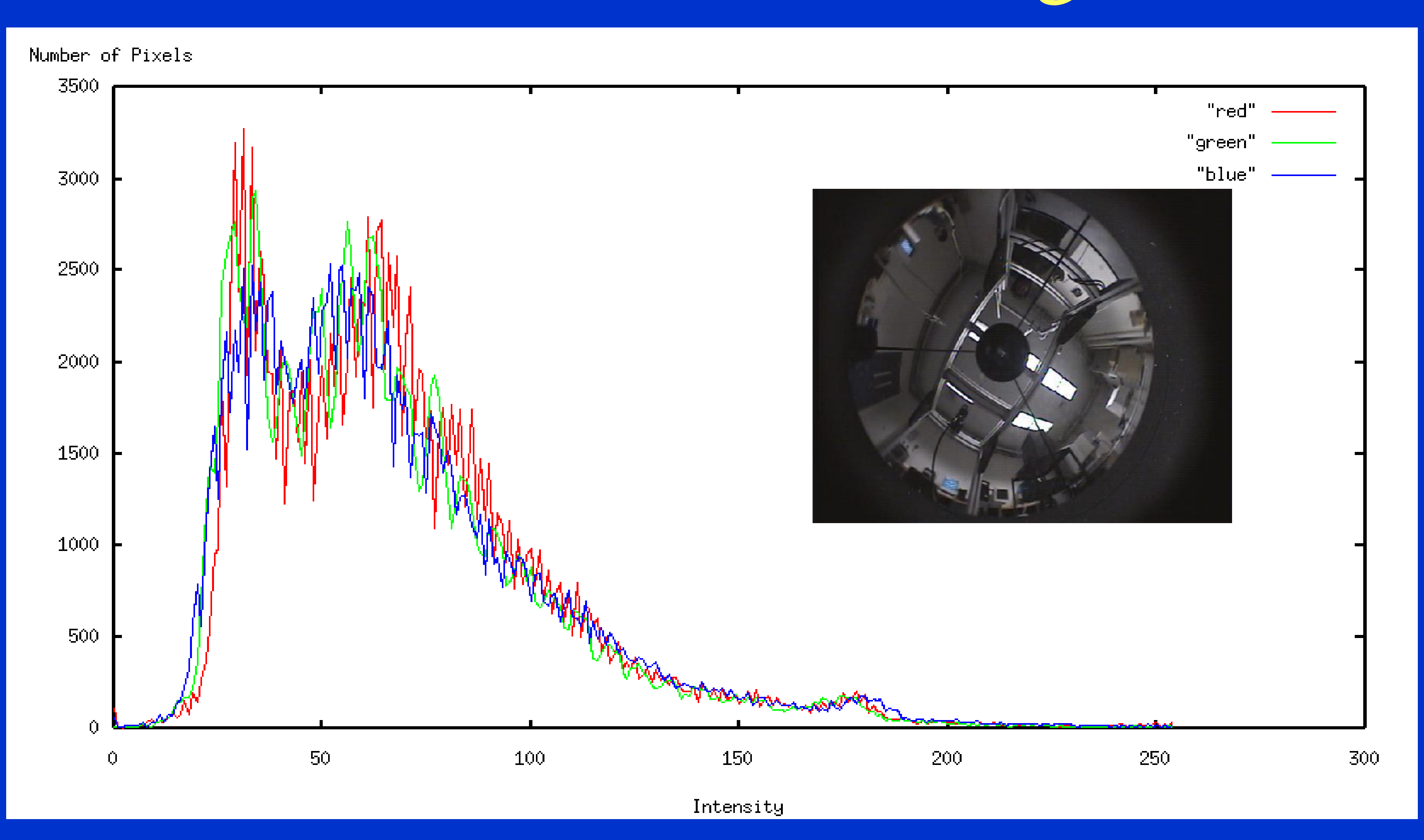
video

Environmental Effects

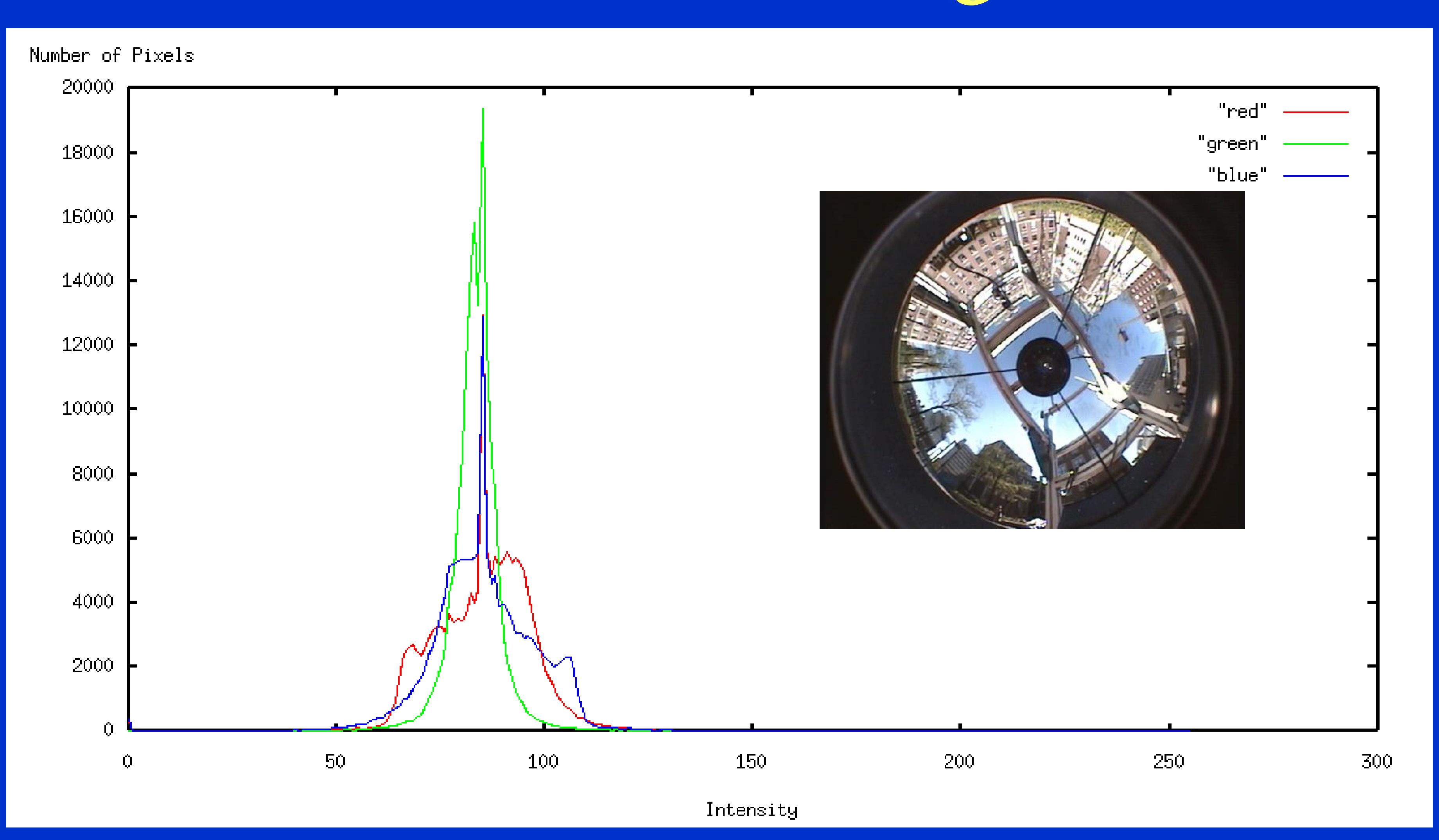
- Indoor environments
  - Controlled lighting conditions
  - No histogram adjustments
- Outdoor environments
  - Large variations in lighting
  - Use a histogram normalization with the percentage of color at each pixel:

$$\frac{R}{R+G+B}, \frac{G}{R+G+B}, \frac{B}{R+G+B}$$

Indoor Image Non-Normalized Histogram



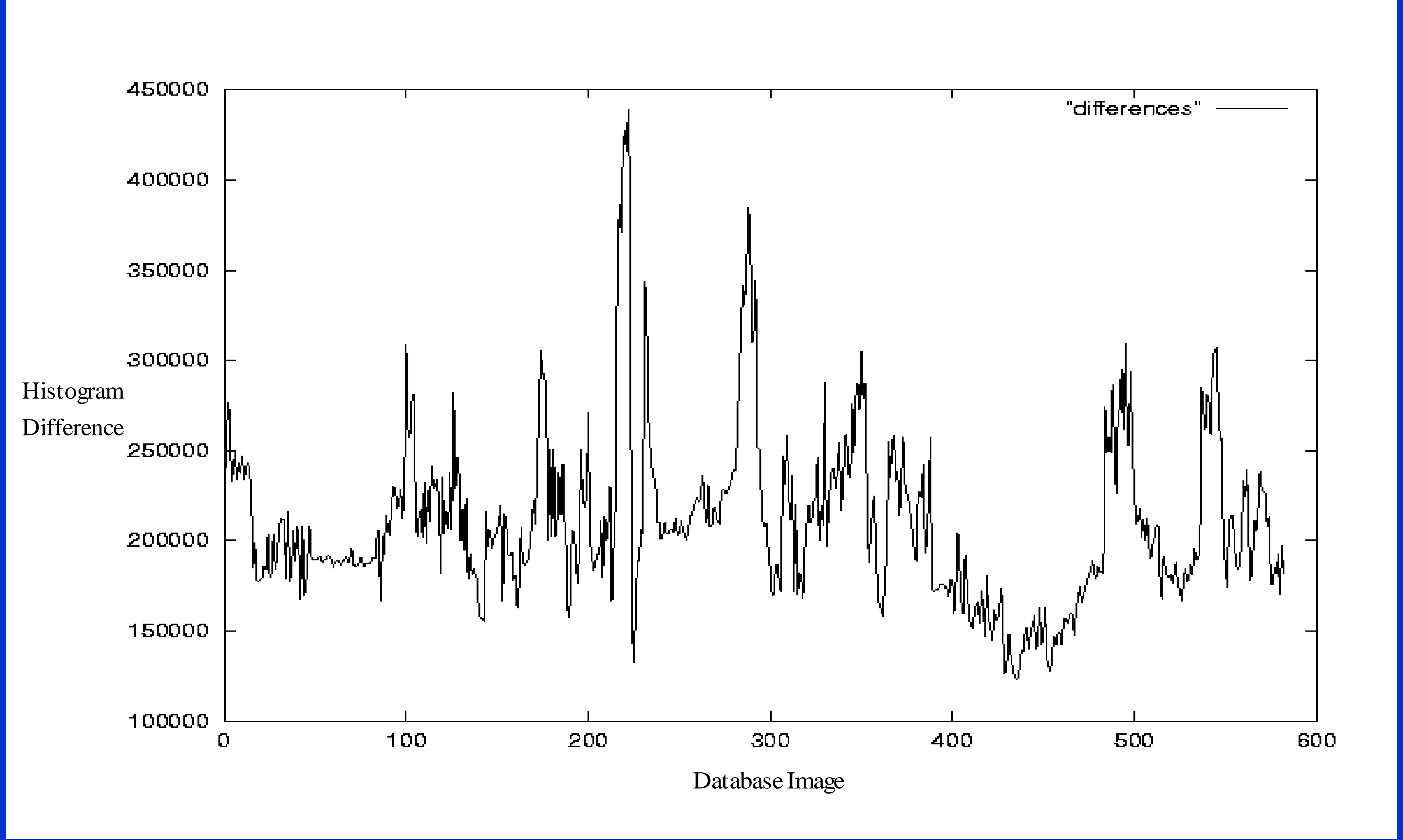
Outdoor Image Normalized Histogram



Matching an Unknown Image with the Database

- Compare unknown image to each image in the database.
- Initially we treat each band (r, g, b) separately.
- The difference between two histograms is the sum of the absolute differences of each bucket.
- Better to sum the differences for all three bands into a single metric than to treat each band separately.
- The region of the database image with the smallest difference is the selected region for this unknown.

Image Matching to the Database



The Test Environments

Indoor
(Hallway View)

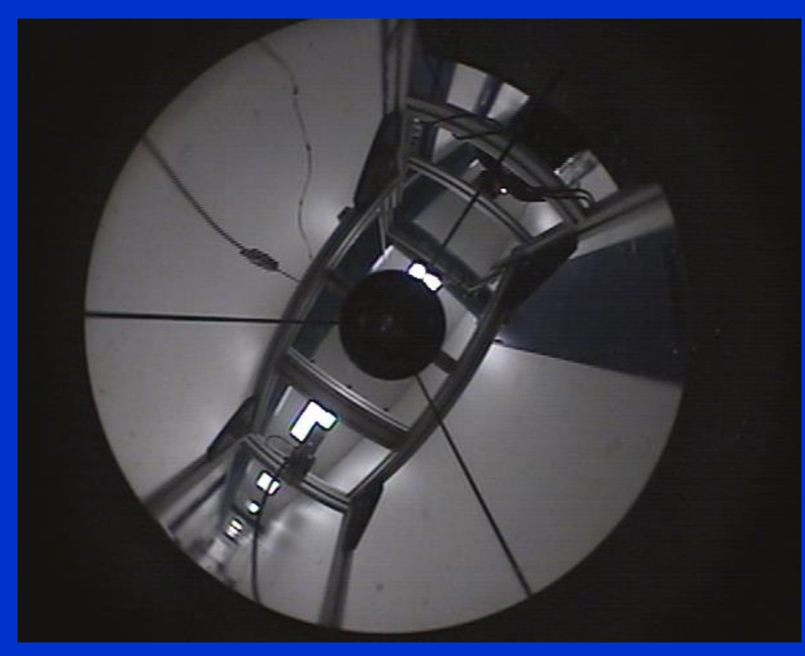
Outdoor
(Aerial View)

Outdoor
(Campus Map)

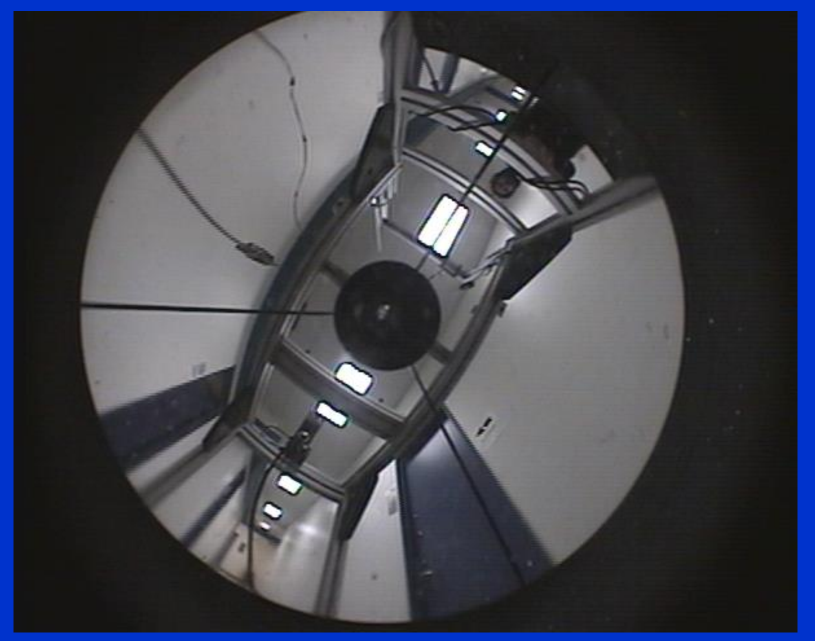
Indoor Results

Region	Images Tested	Non-Normalized % Correct	Normalized % Correct
1	21	100%	95%
2	12	83%	92%
3	9	77%	89%
4	5	20%	20%
5	23	74%	91%
6	9	89%	78%
7	5	0%	20%
8	5	100%	40%
Total	89	78%	80%

Ambiguous Regions



South
Hallway

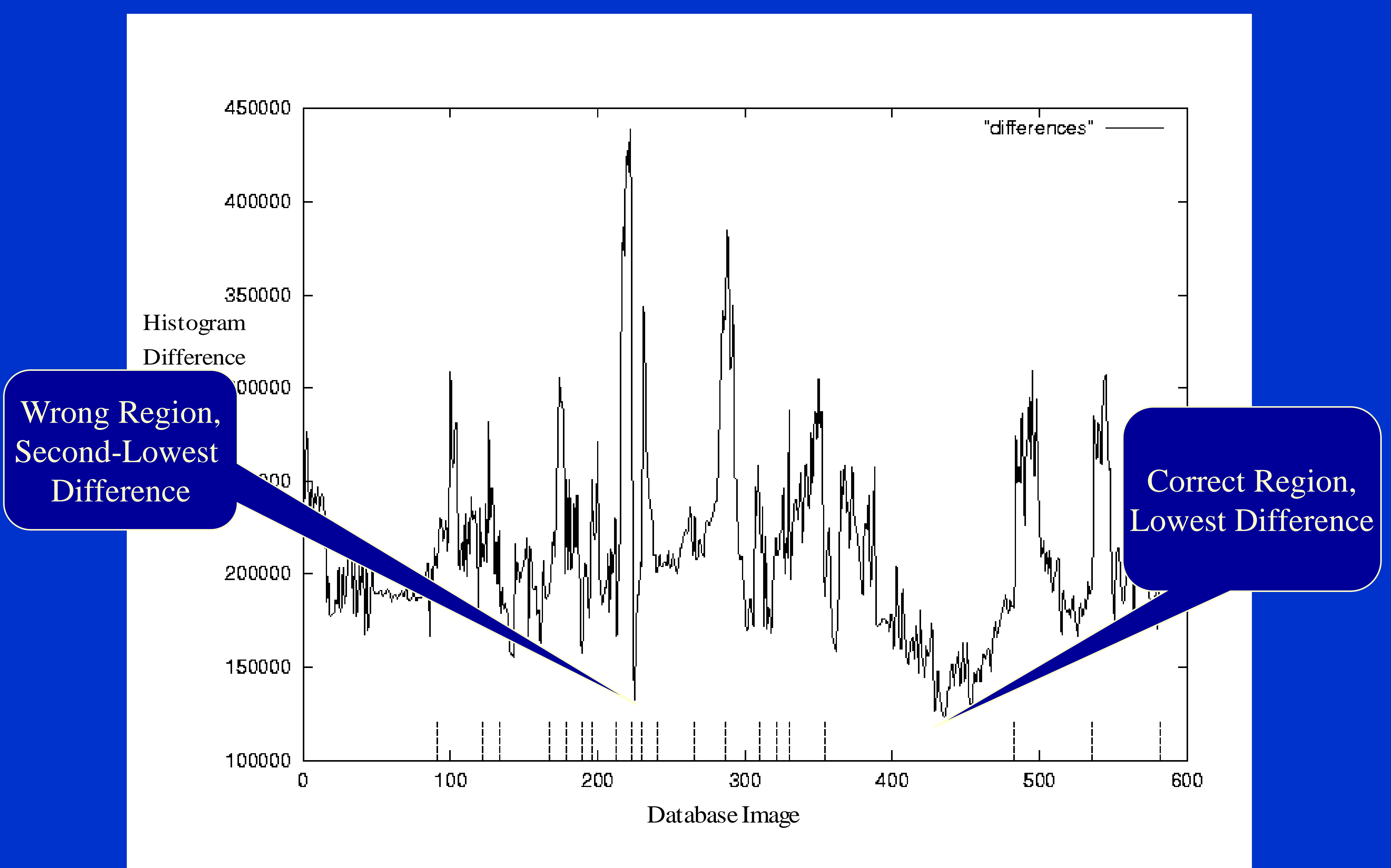


North
Hallway

Outdoor Results

Region	Images Tested	Non-Normalized % Correct	Normalized % Correct
1	50	58%	95%
2	50	11%	39%
3	50	29%	71%
4	50	25%	62%
5	50	49%	55%
6	50	30%	57%
7	50	28%	61%
8	50	41%	78%
Total	400	34%	65%

The Best Candidate Regions



Conclusions

- In 80% of the cases we were able to narrow down the robot's location to only 2 or 3 possible regions without any prior knowledge of the robot's position.
- Our goal was to reduce the number of possible models that the fine-grained visual localization method needed to examine.
- Our method effectively quartered the number of regions that the fine-grained method had to test.

Future Work

- What is needed is a fast secondary discriminator to distinguish between the 2 or 3 possible regions.
- Histograms are limited in nature because of their total reliance on the color of the scene.
- To counter this we want to incorporate more geometric data into our database, such as edge images.