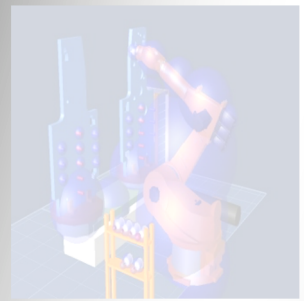
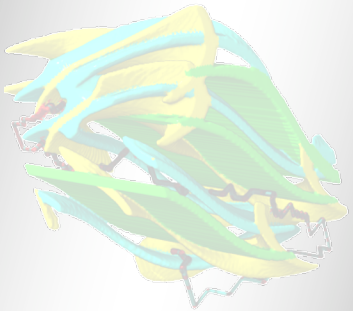
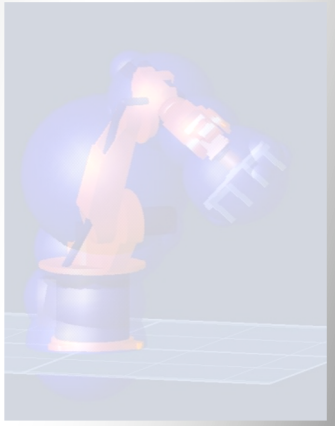
Lazy-PRM







Why do all this extensive collision testing?

(R. Bohlin and L. Kavraki)

Disadvantages ..... so far!

- Proposed algorithms spend a lot of time planning paths that will never get used.
  - Especially, when you just want to do single-queries
- Heavily reliant on fast collision checking / distance computation.
- An attempt to solve these is made with Lazy PRMs
  - Tries to minimize collision checks
  - Tries to reuse information gathered by queries

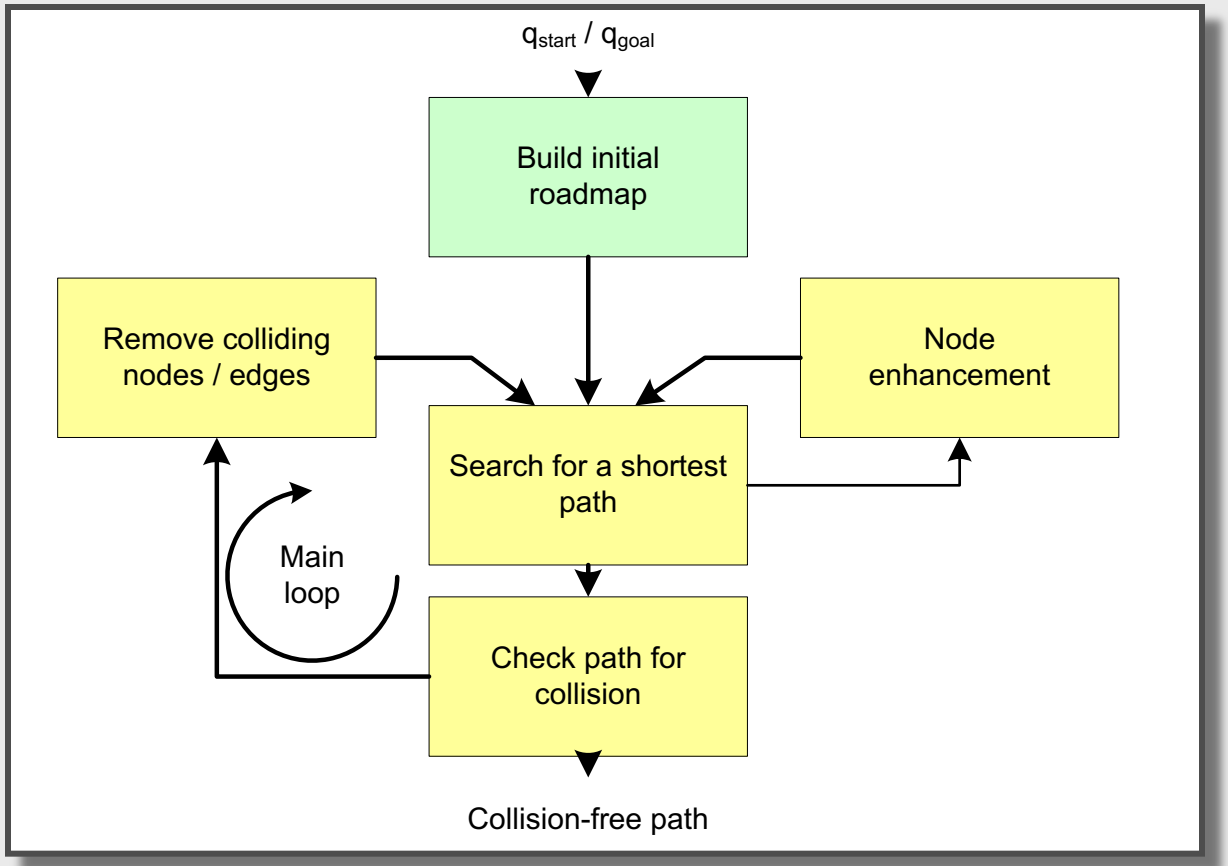
References

Ideas taken from presentation & papers:

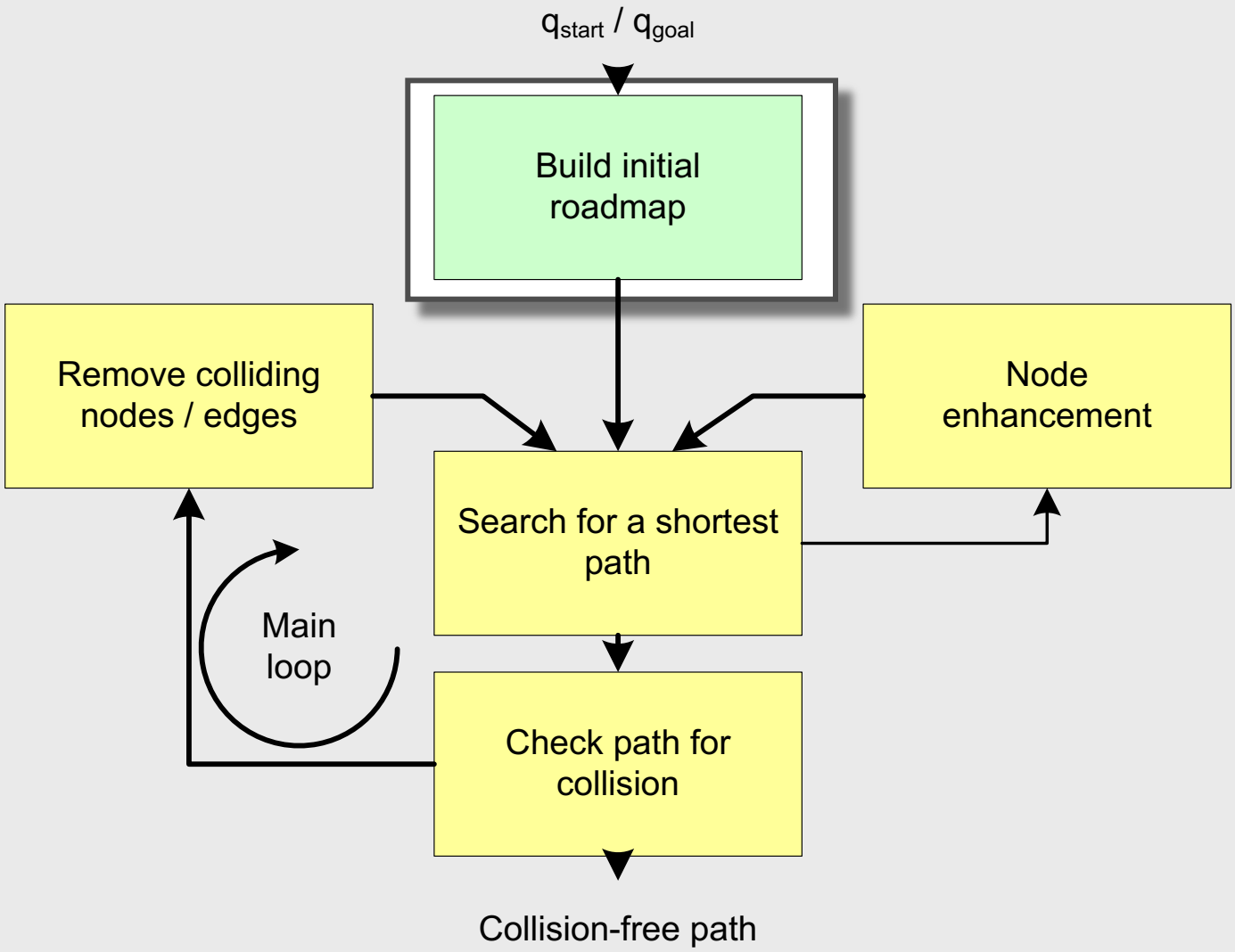
- **"On Delaying Collision Checking in PRM Planning"**: presentation (G. Sanchez, J. Latombe, Computer Science Department, Stanford University)
- **"Path Planning using Lazy-PRM"**: paper of T.V.N. Sri Ram
- **"A General Framework for Sampling on the Medial Axis of the Free Space"**: presentation (Jyh-Ming Lien, Shawna Thomas, Nancy Amato)
- **"The Gaussion Sampling Strategy for Probabilistic Roadmap Planners"**: paper (Boor, Overmars, Stappen)
- **"Probabilistic Motion Planning"**: presentation (Shai Hirsch)
- "**Rapidly-Exploring Random Trees**": presentation (Steven Lavalle)

Lazy-PRM: the idea behind

- Instead of building a roadmap of feasible paths, **build a roadmap of paths assumed to be feasible**.
  - Test feasibility as late as possible → during query phase

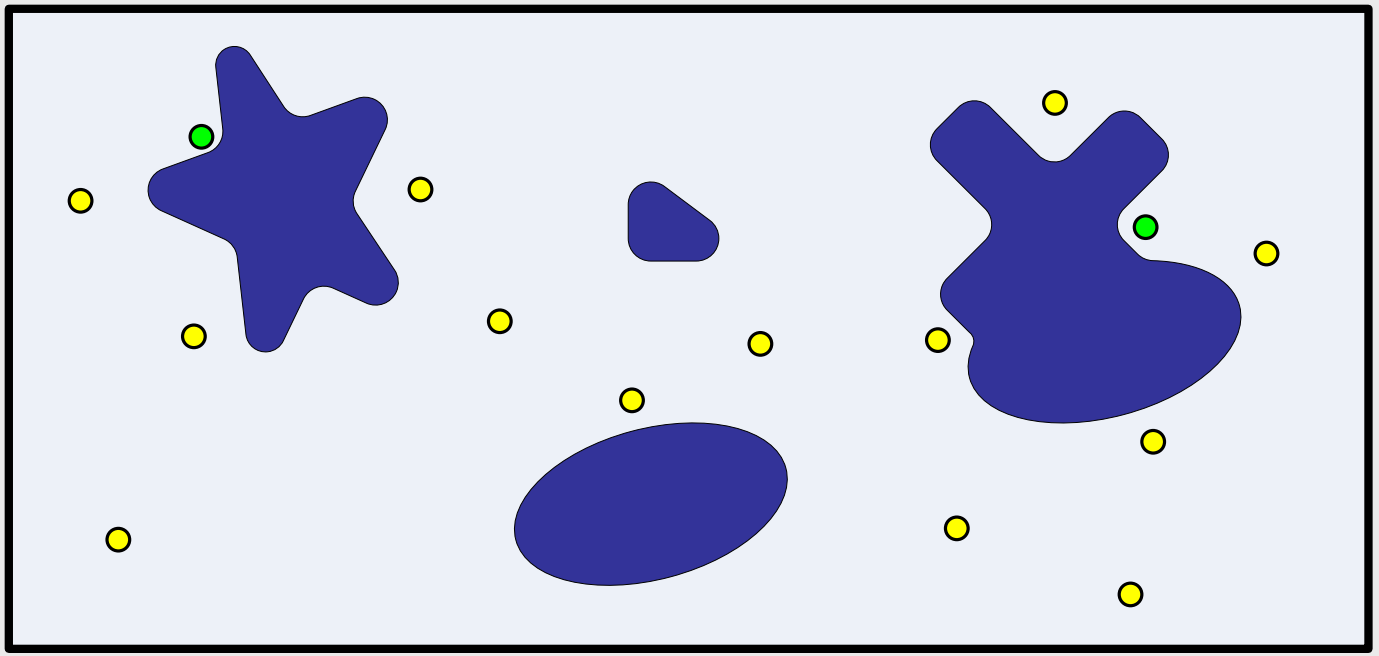


Lazy-PRM: The flow.....



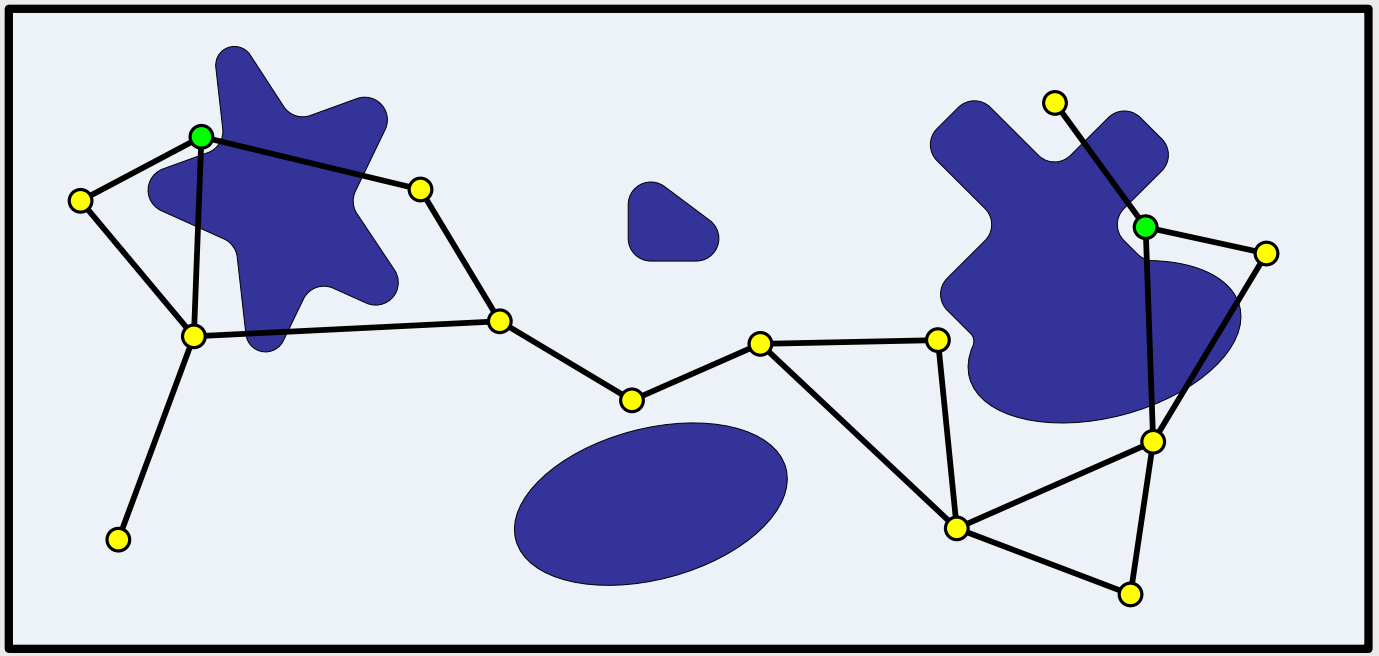
Lazy-PRM: Build initial Roadmap

- Insert start & goal and N_{init} uniformly distributed nodes, **don't care about collisions**

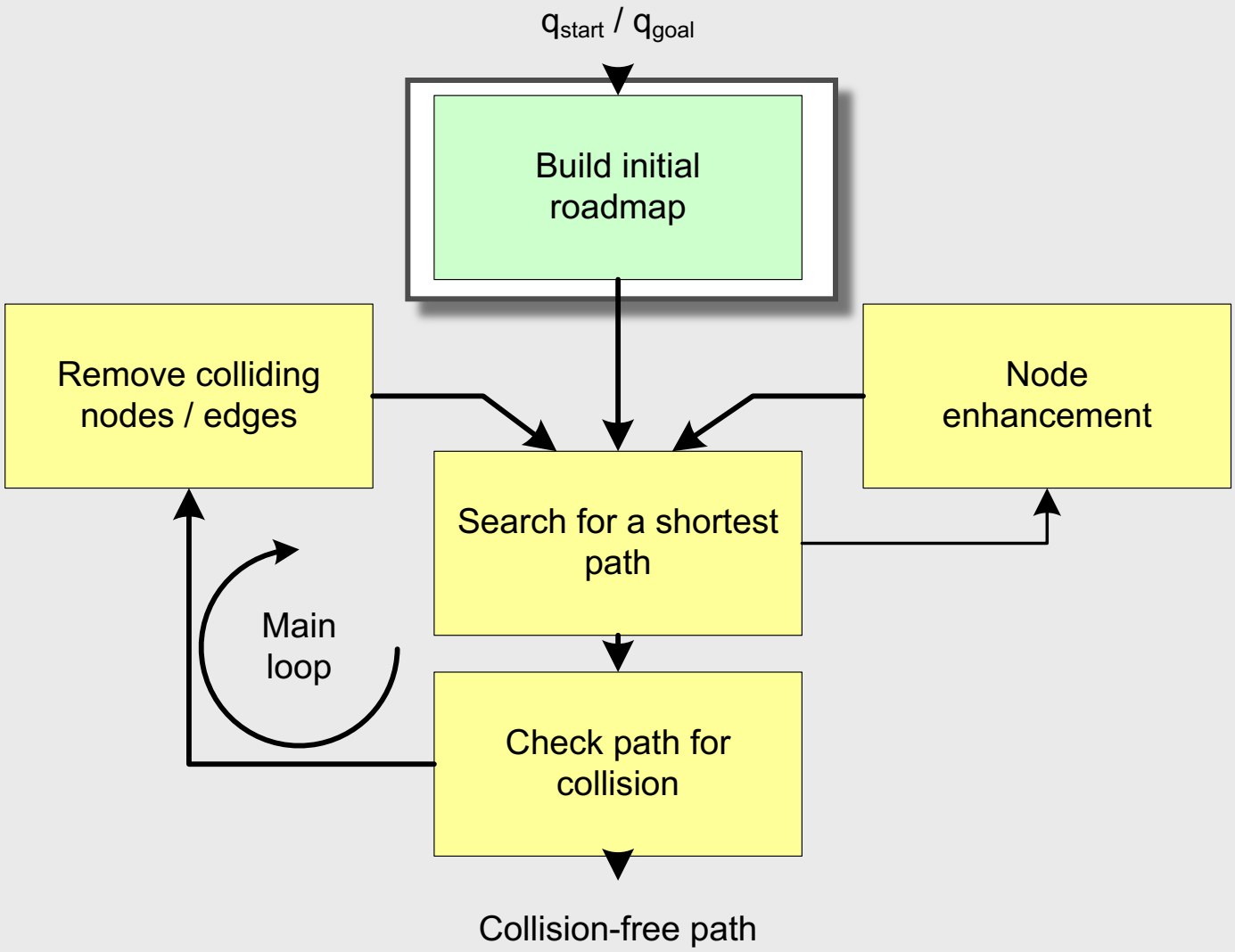


Lazy-PRM: Build initial Roadmap

- Insert start & goal and N_{init} uniformly distributed nodes, **don‘t care about collisions**
- Build roadmap using k-closest strategy or something similar, **don‘t care about collisions**

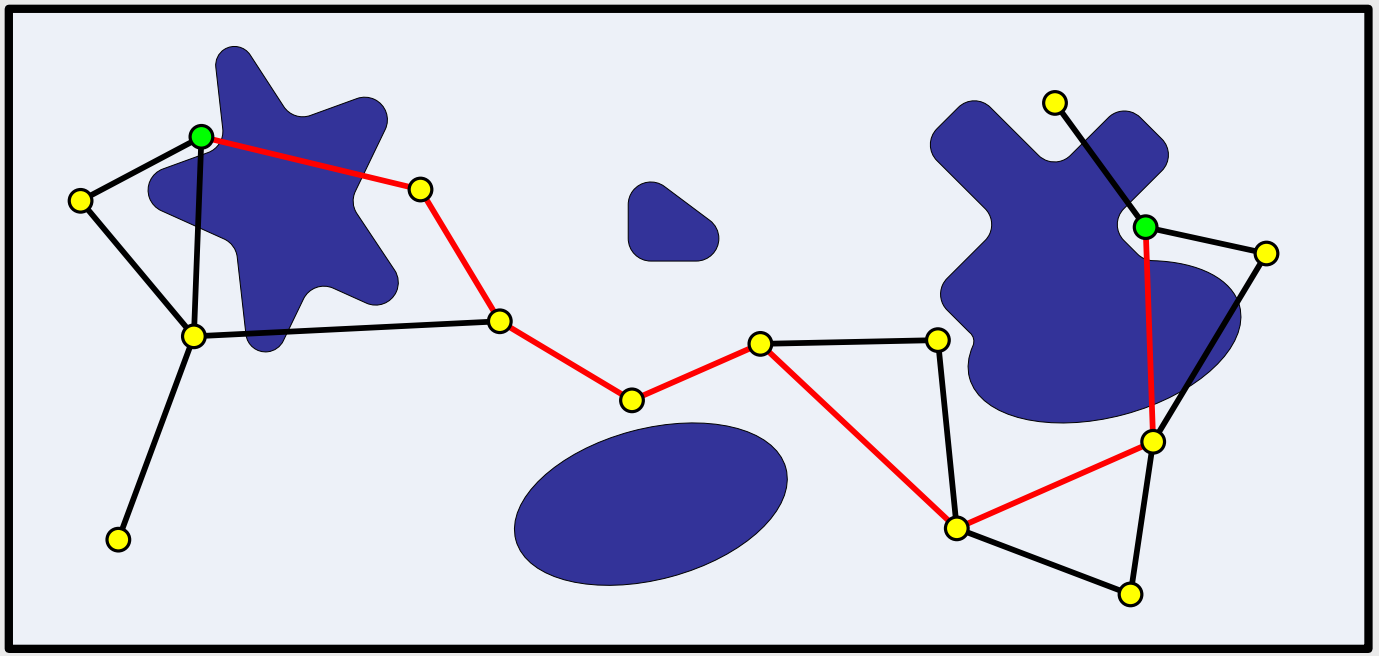


Lazy-PRM: The flow.....

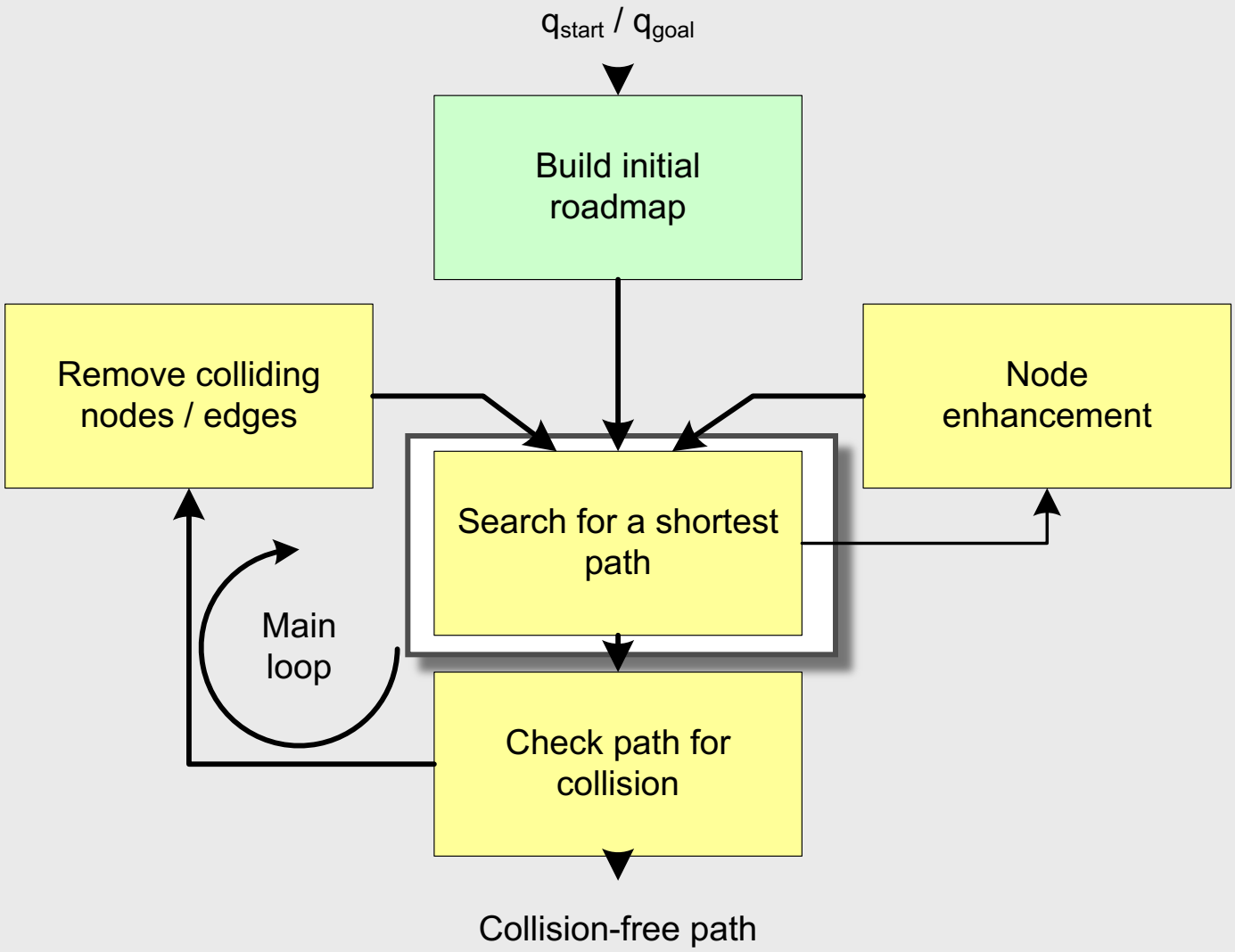


Lazy-PRM: Search shortest path

- Search shortest path on computed Roadmap, **don‘t care about collisions**.
  - Using A* or Dijkstra

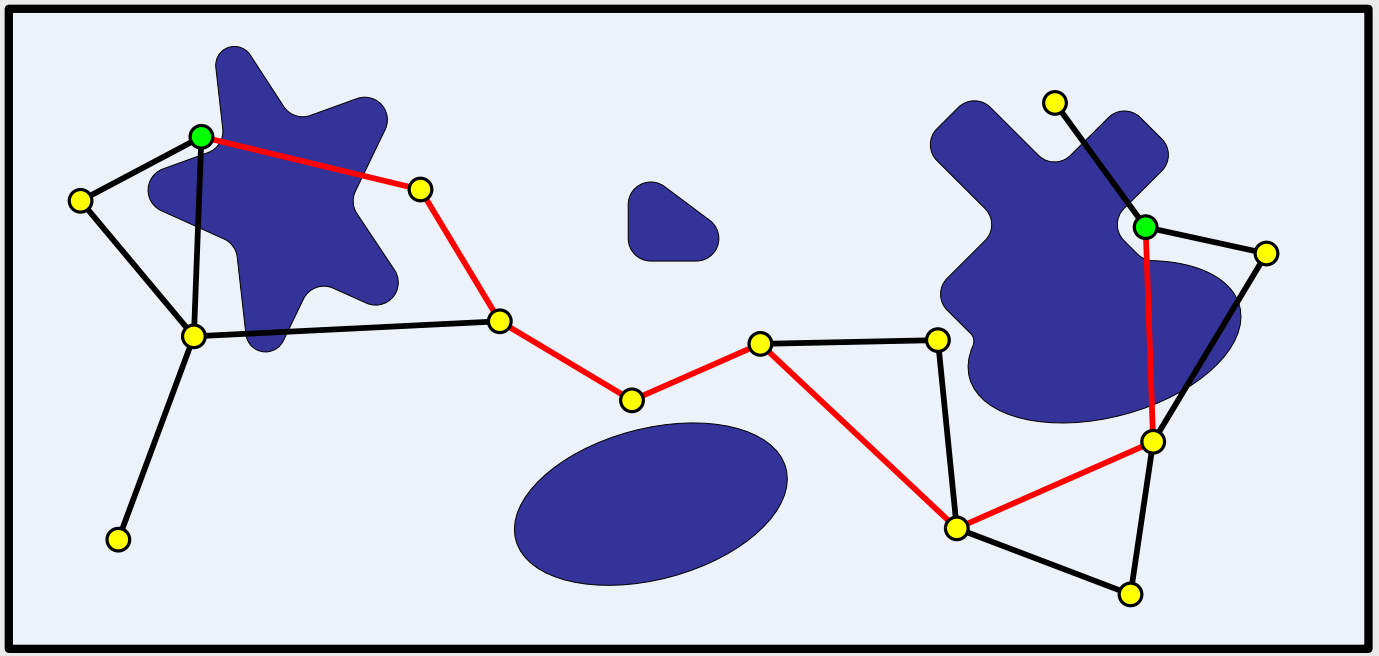


Lazy-PRM: The flow.....



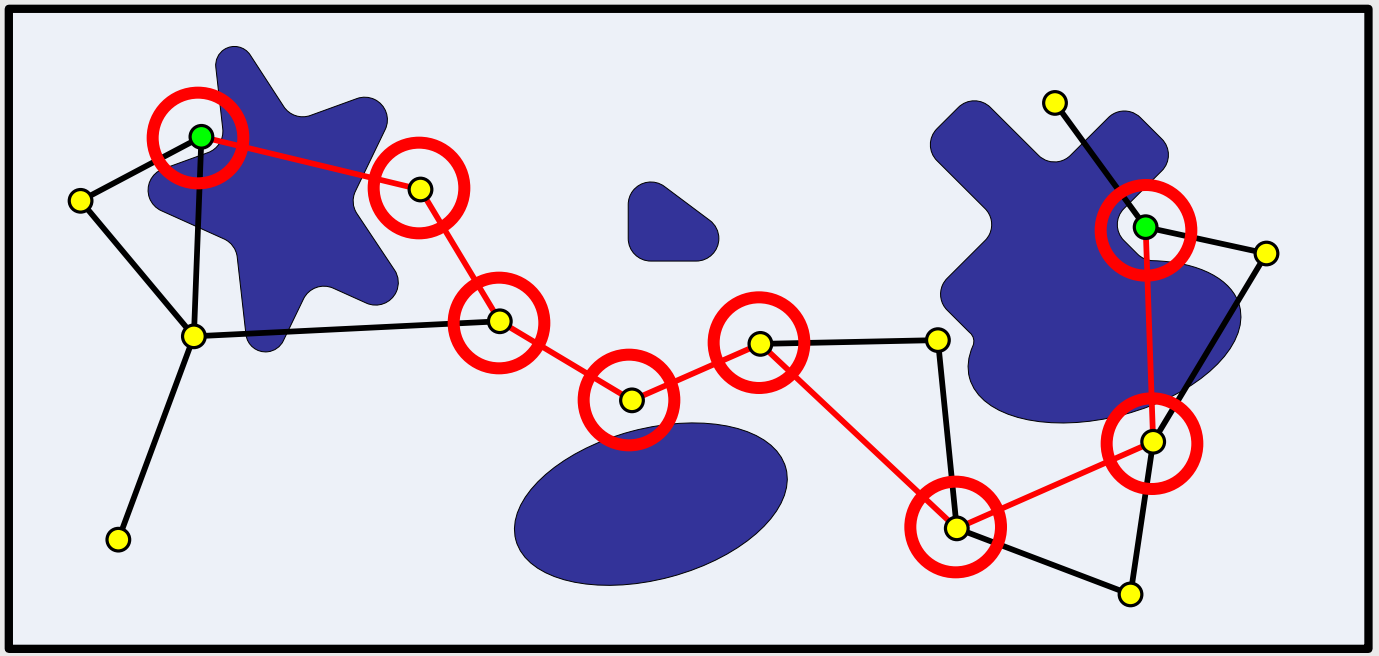
Lazy-PRM: Check nodes /path for collisions

- Check nodes for collisions



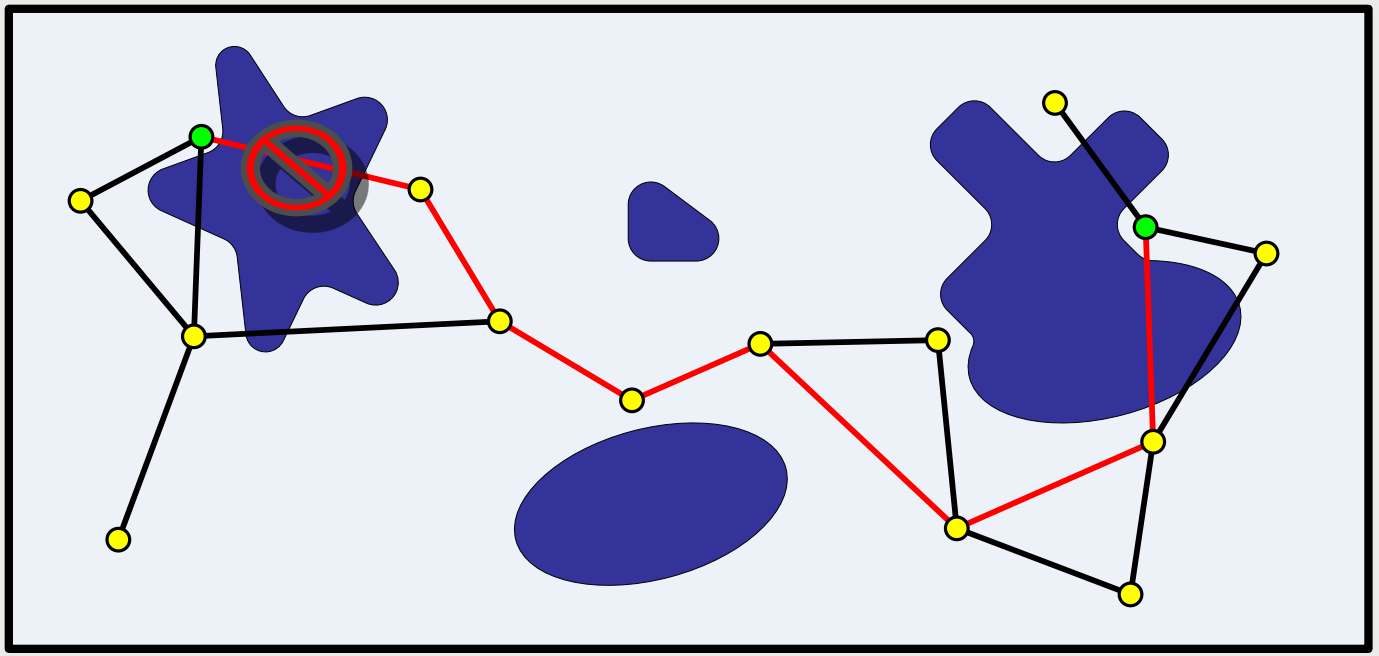
Lazy-PRM: Check nodes /path for collisions

- Check nodes for collisions
  - Starting with first and last point of path
  - Check nodes alternatingly
  - If a node is colliding remove node and compute again shortest path

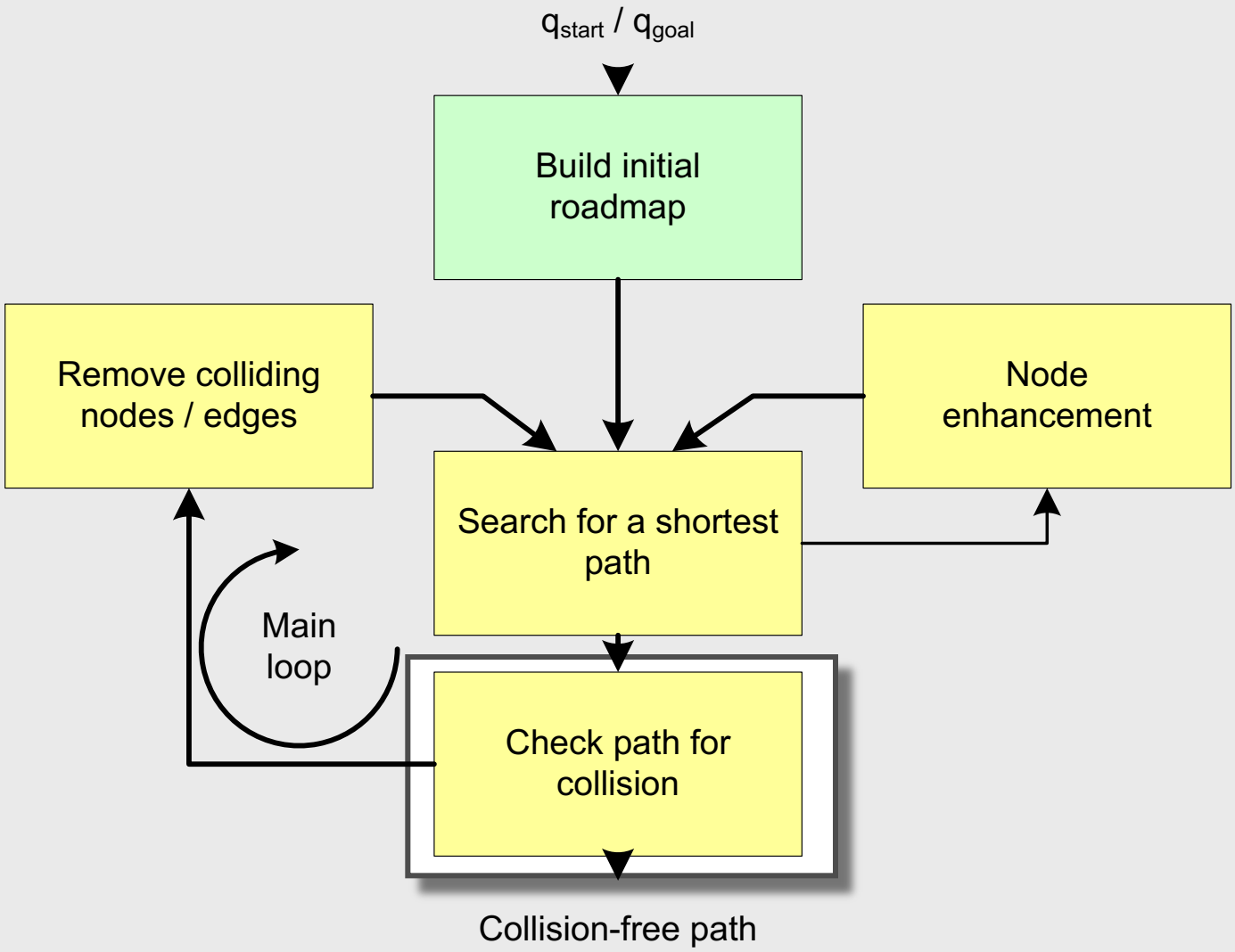


Lazy-PRM: Check nodes /path for collisions

- Check nodes for collisions
- Check edges for collisions
  - Alternatingly like nodes

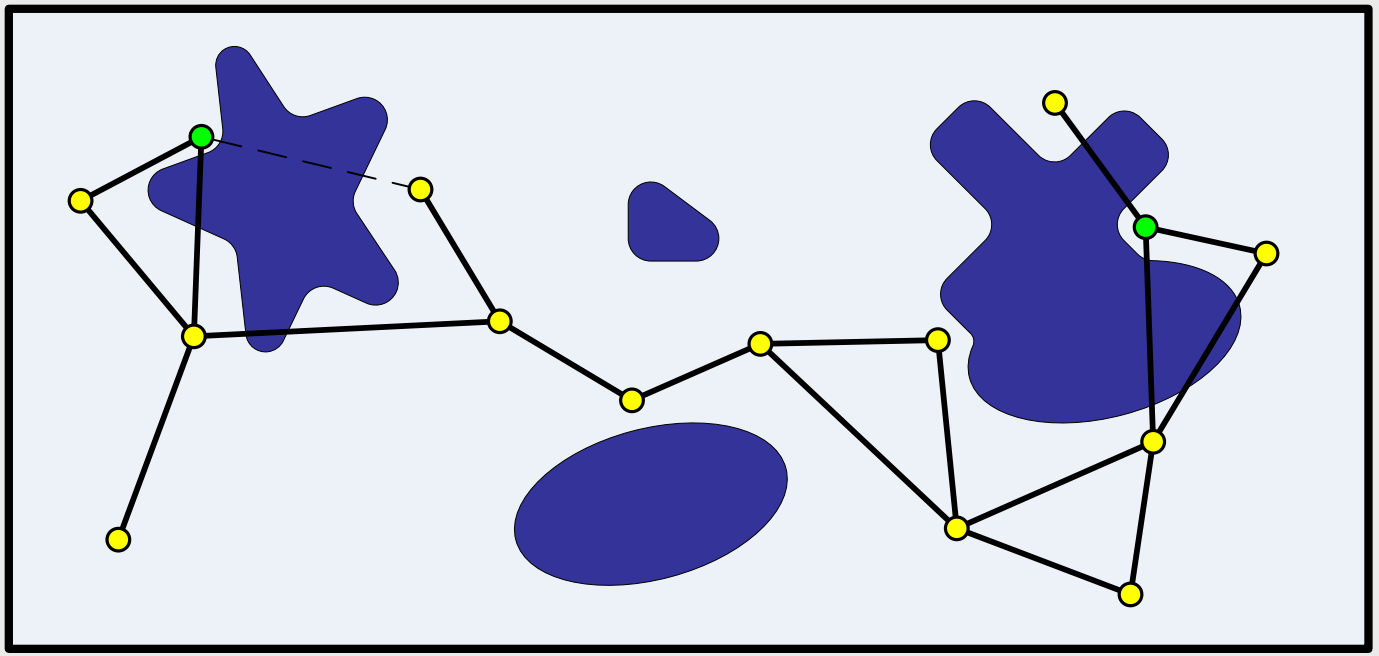


Lazy-PRM: The flow.....

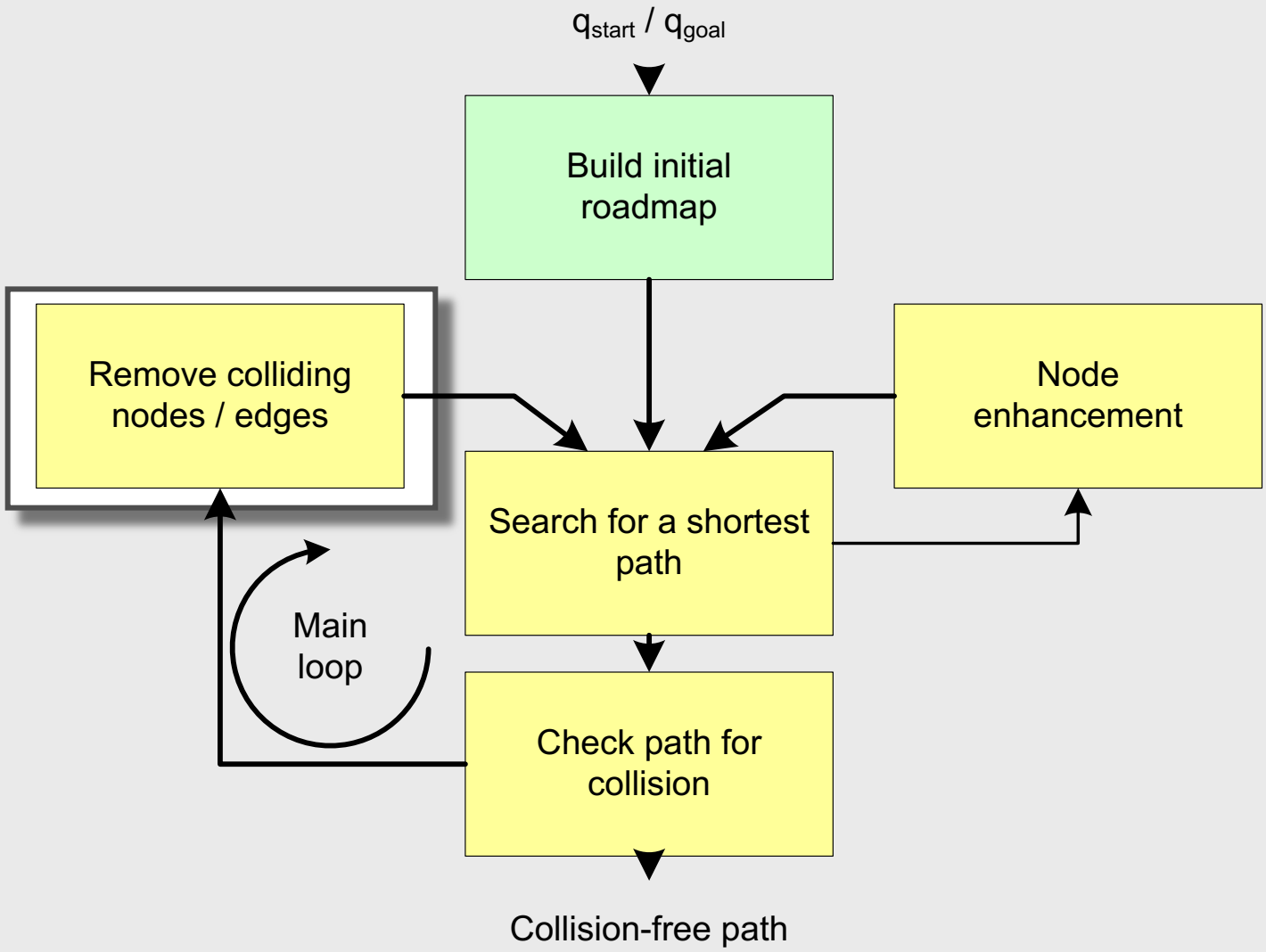


Lazy-PRM: Check nodes /path for collisions

- Check nodes for collisions
- Check edges for collisions
  - Remove colliding edges

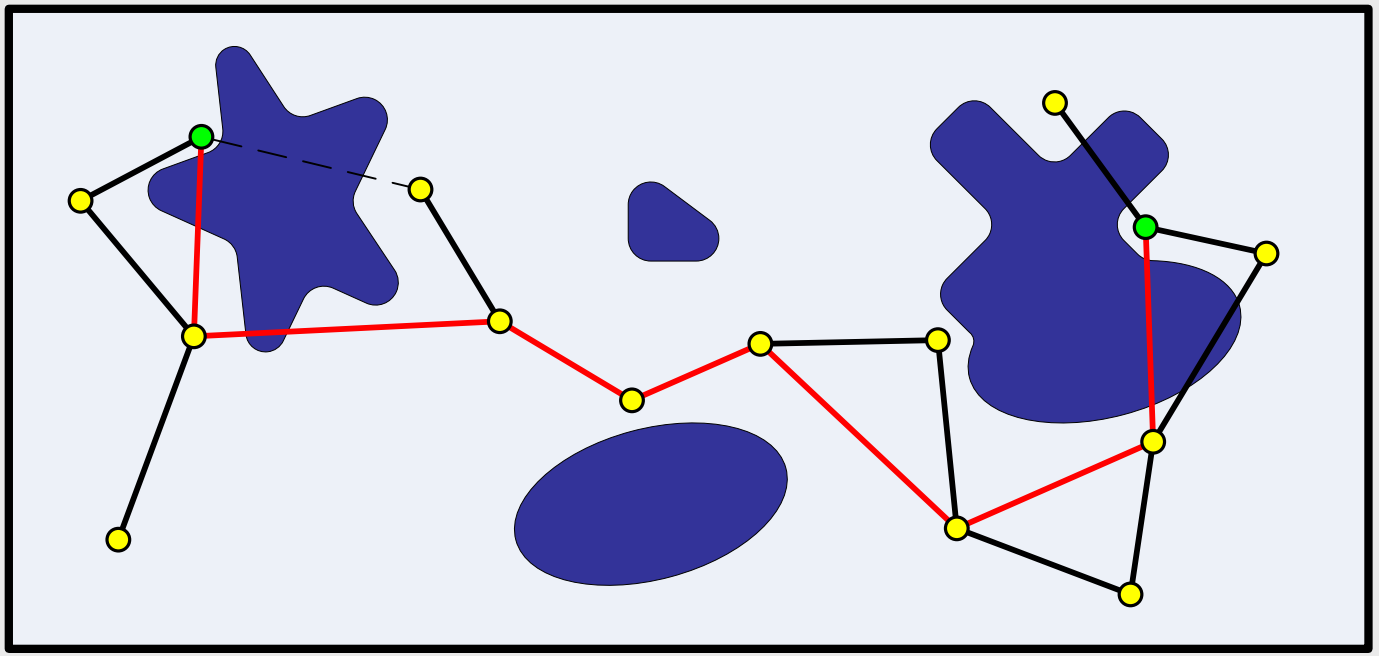


Lazy-PRM: The flow.....



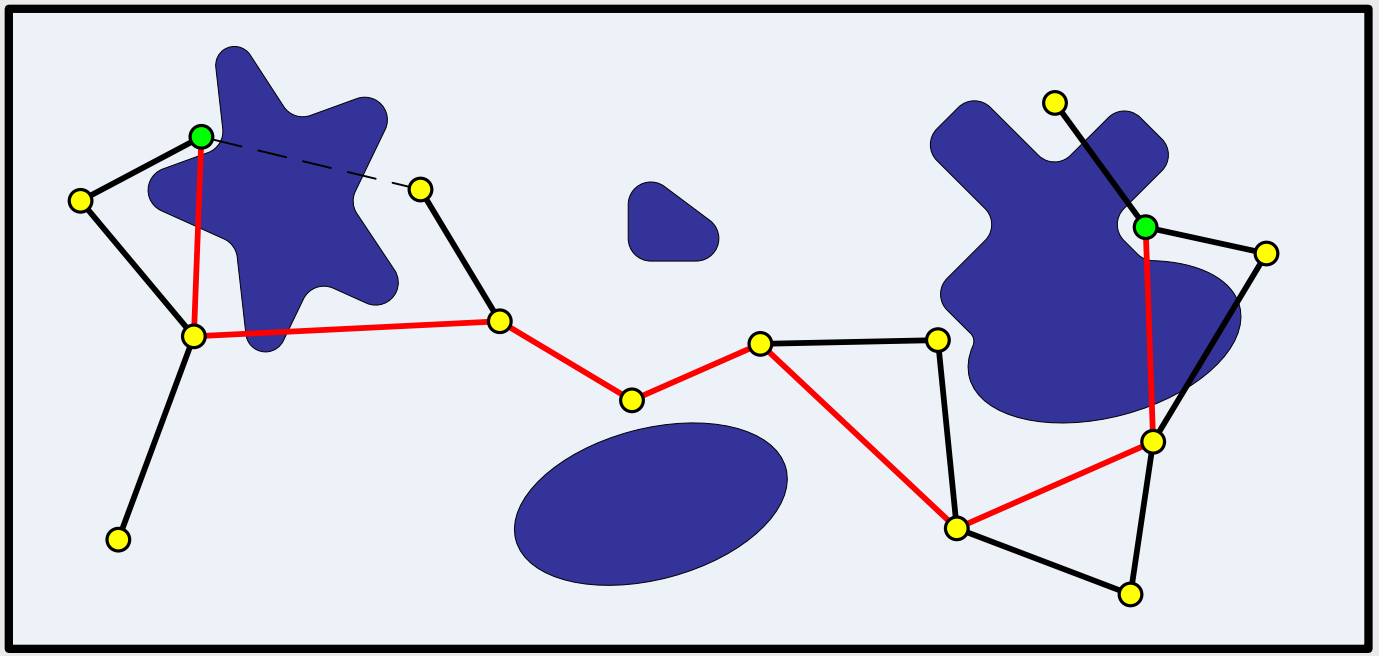
Lazy-PRM: Search again shortest path

- Search shortest path on computed Roadmap, **don‘t care about collisions**.
  - Using A* or Dijkstra



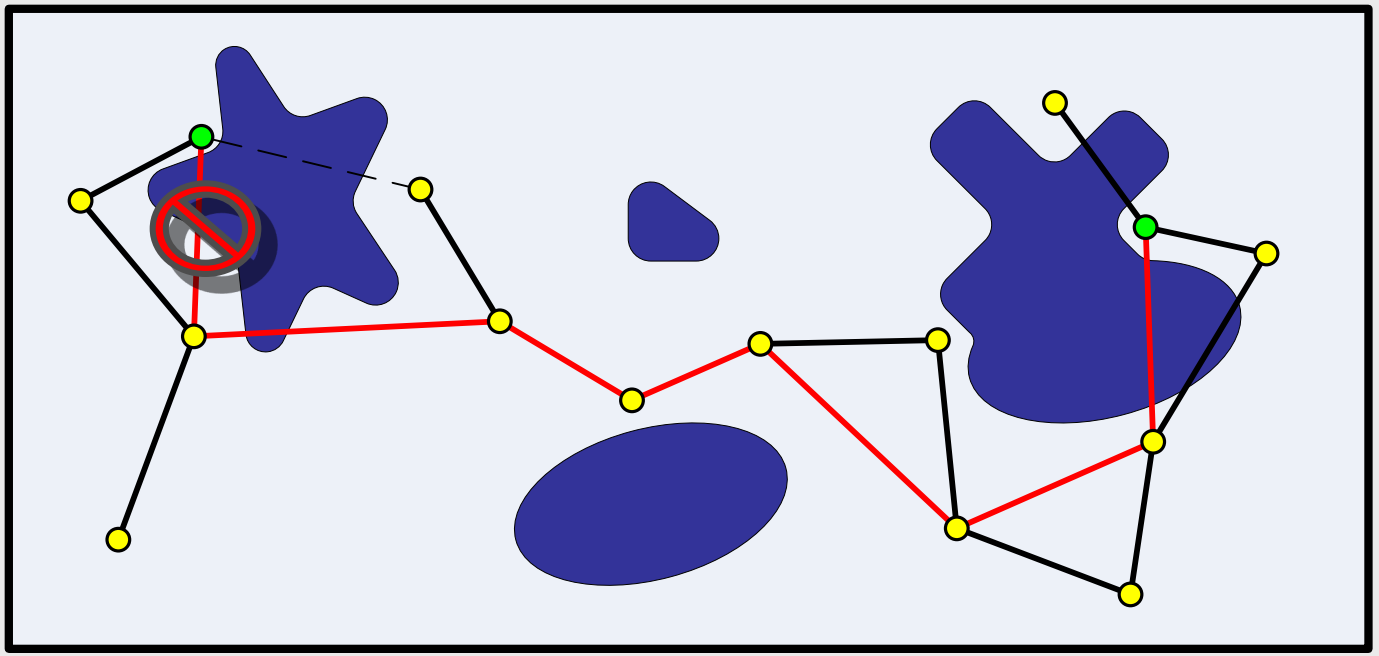
Lazy-PRM: Check nodes /path for collisions

- Check nodes for collisions



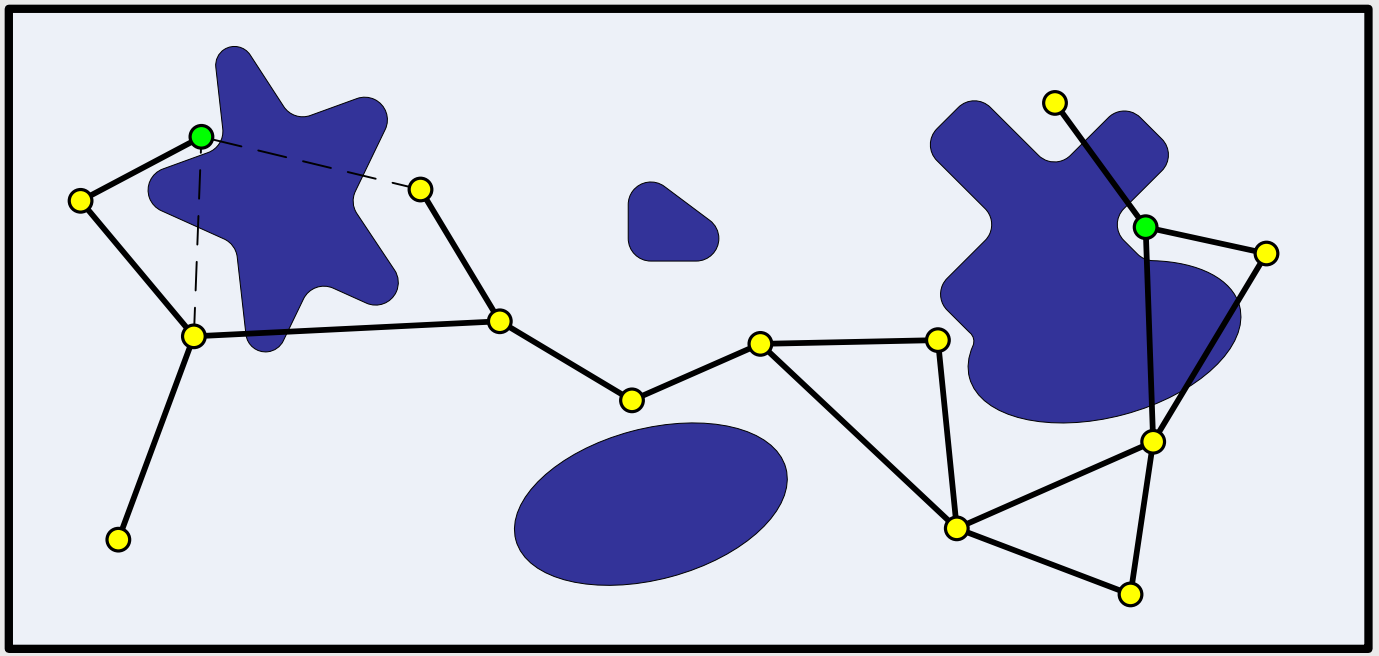
Lazy-PRM: Check nodes /path for collisions

- Check nodes for collisions
- Check edges for collisions



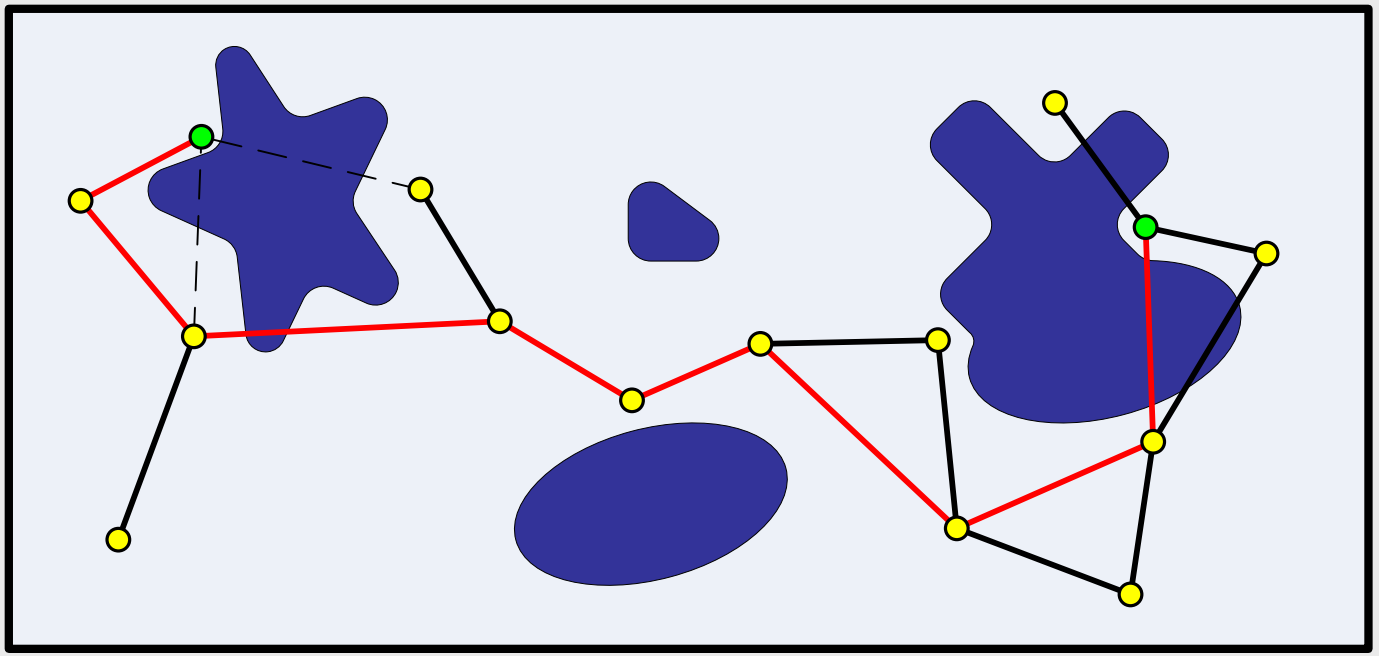
Lazy-PRM: Check nodes /path for collisions

- Check nodes for collisions
- Check edges for collisions
  - Remove colliding edges



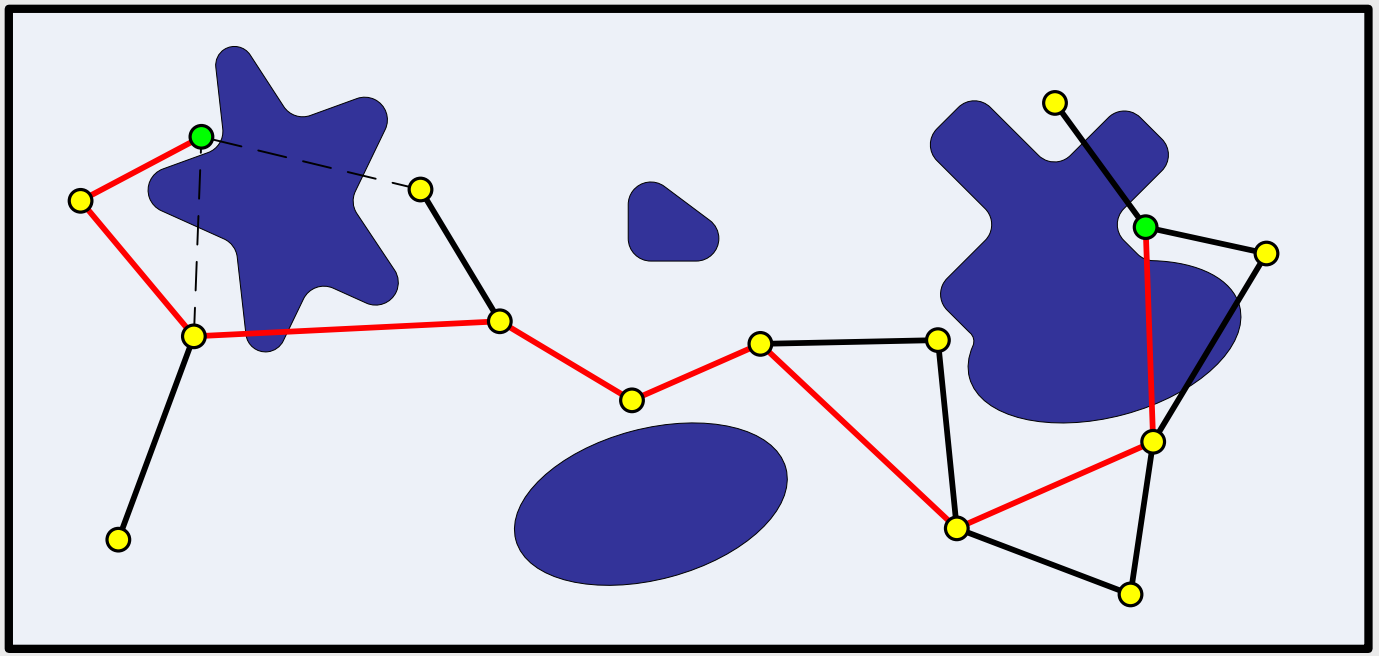
Lazy-PRM: Search again shortest path

- Search shortest path on computed Roadmap, **don‘t care about collisions**.
  - Using A* or Dijkstra



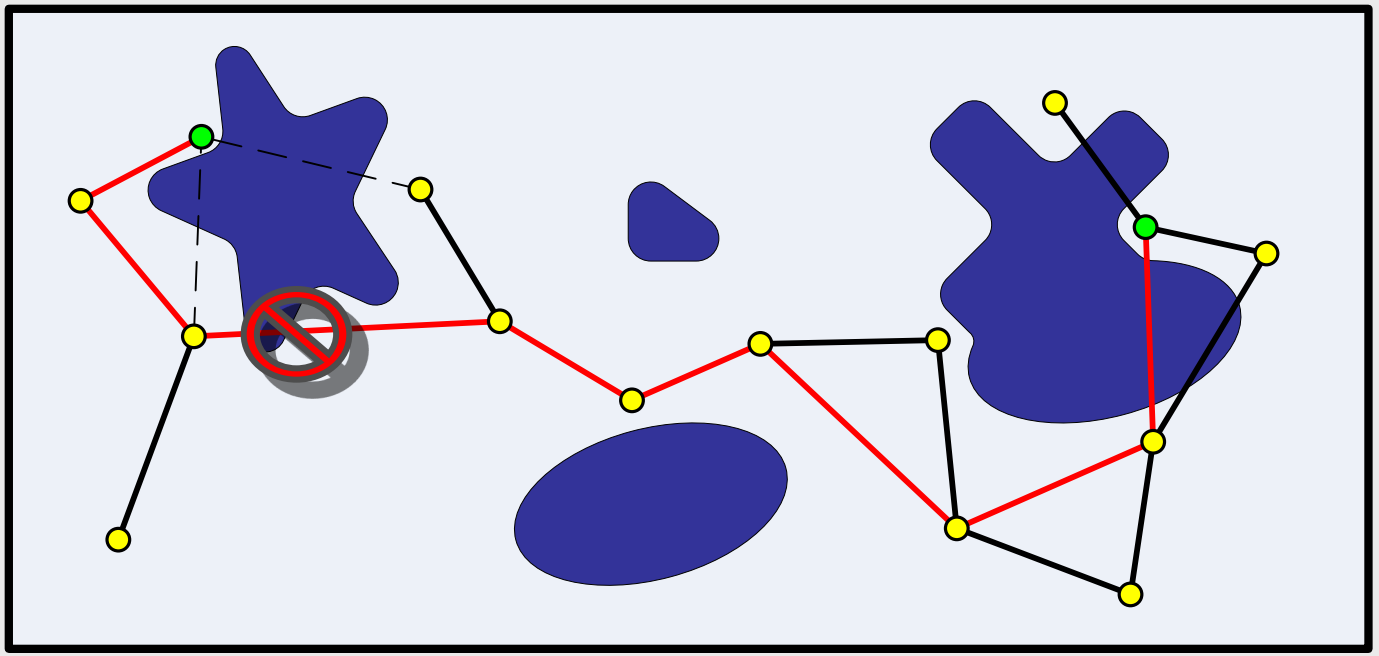
Lazy-PRM: Check nodes /path for collisions

- Check nodes for collisions



Lazy-PRM: Check nodes /path for collisions

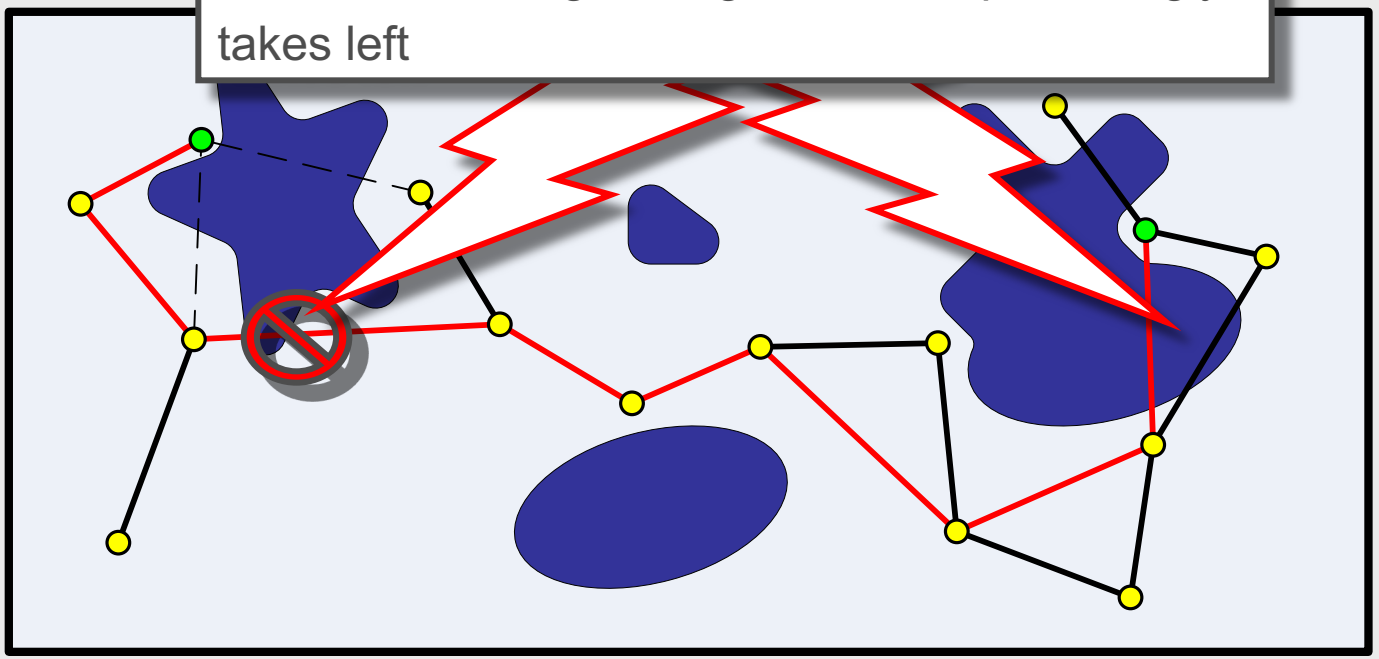
- Check nodes for collisions
- Check edges for collisions



Lazy-PRM: Check nodes /path for collisions

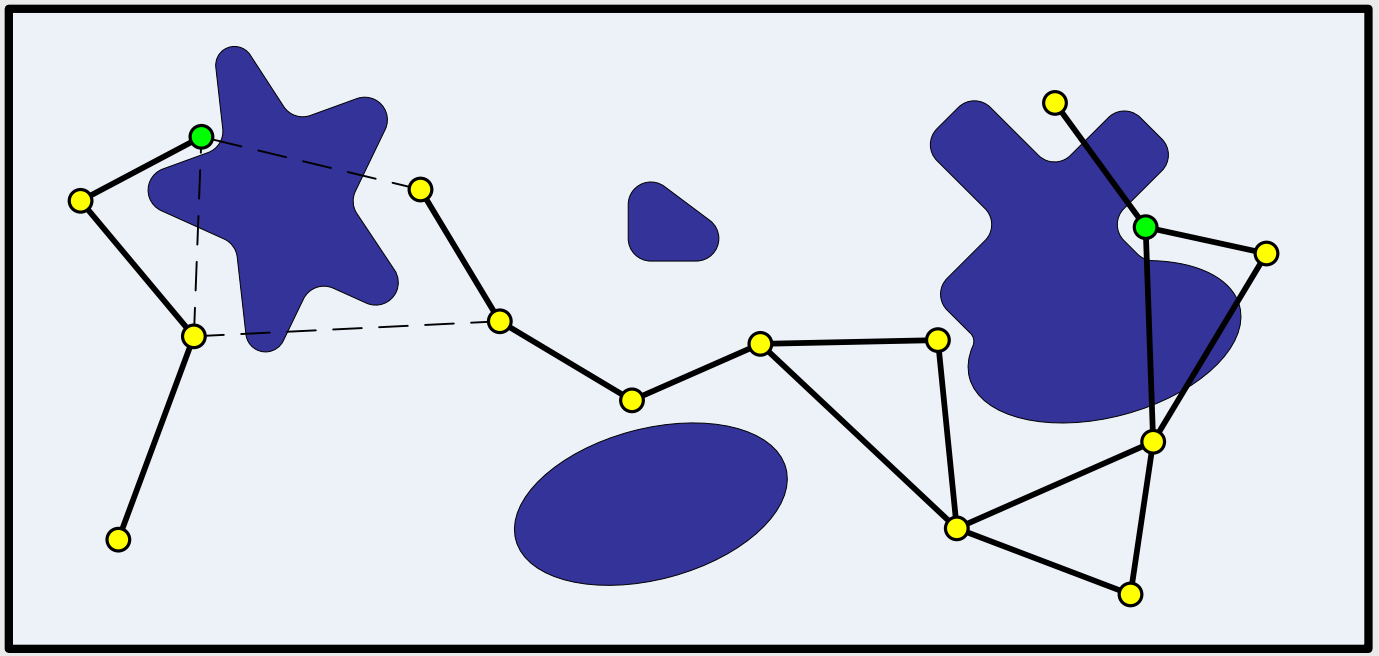
- Check nodes for collisions
- Check edges for collisions

Errata! Following correctly the algorithm right edge would be the first to be found colliding, due to alternating testing ←→ example wrongly takes left



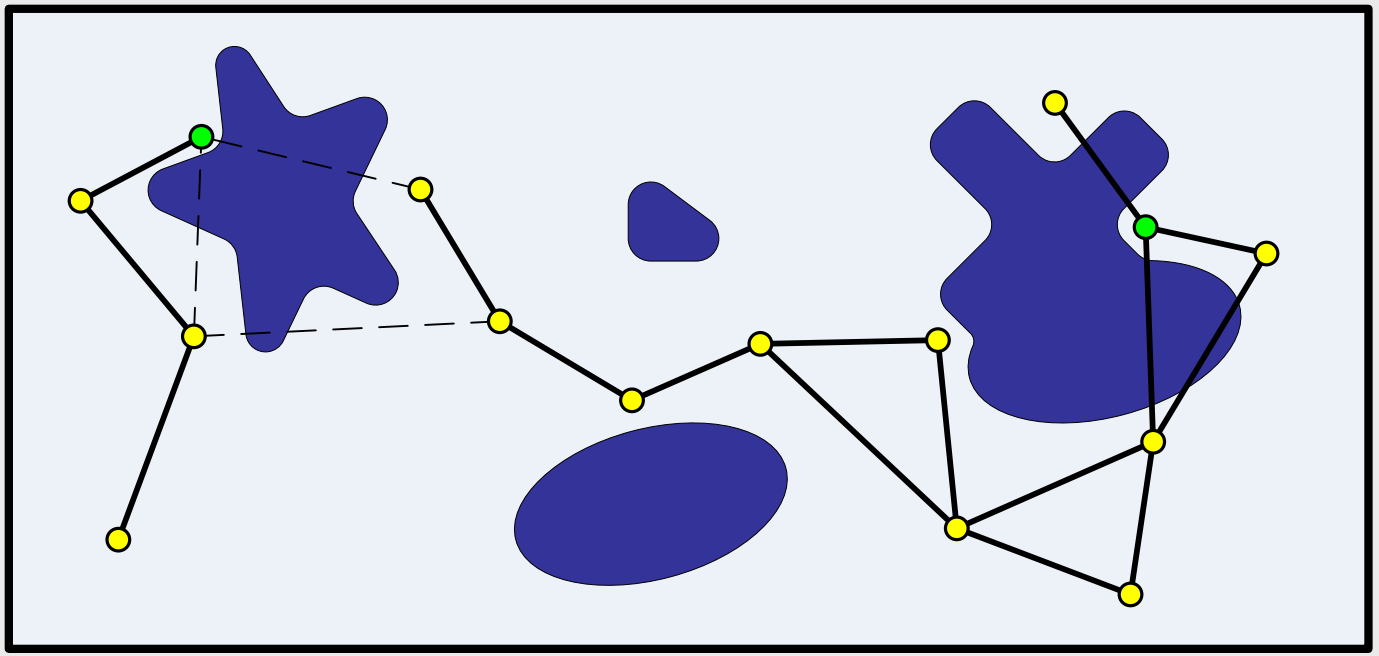
Lazy-PRM: Check nodes /path for collisions

- Check nodes for collisions
- Check edges for collisions
  - Remove colliding edges

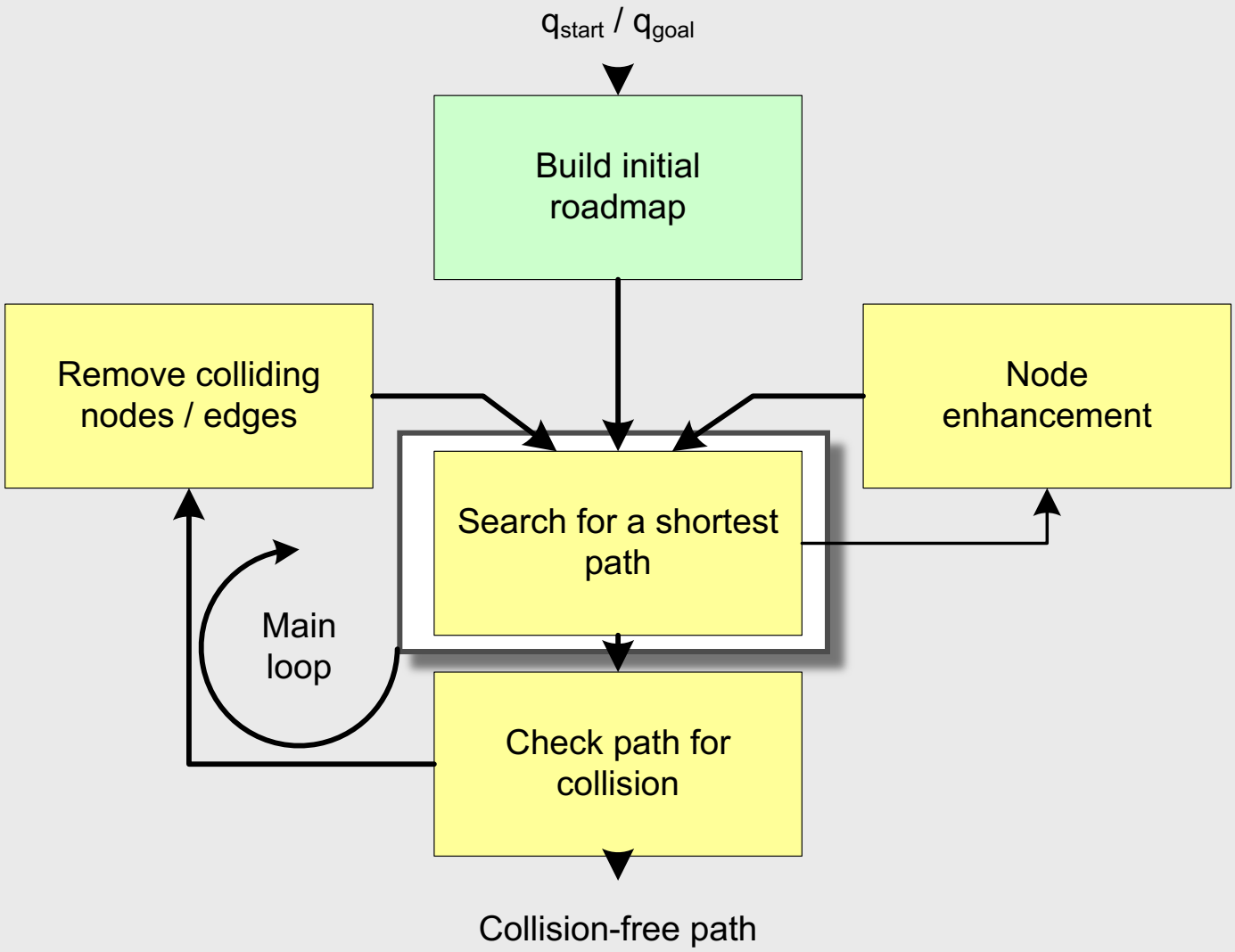


Lazy-PRM: Search again shortest path

- Search shortest path on computed Roadmap, **don‘t care about collisions**.
  - Using A* or Dijkstra
  - **NO path is found**



Lazy-PRM: The flow.....

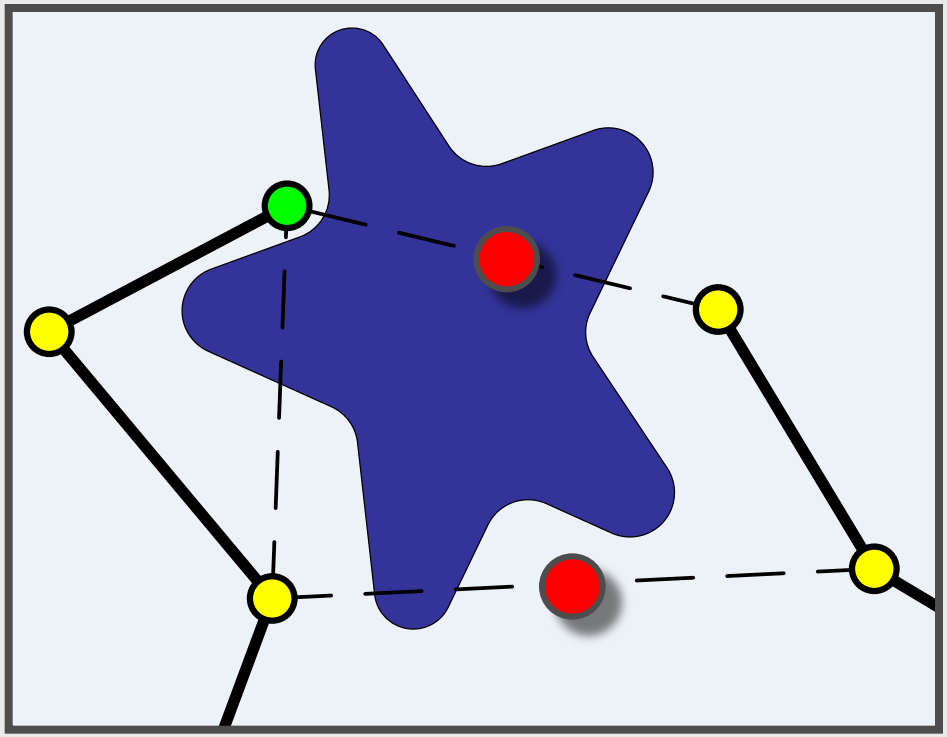


Lazy-PRM: Node enhancement

- If we don‘t create new nodes algorithm fails.
- Strategies:
  - Create again uniformly distributed nodes, or
  - Choose special regions to create nodes: **Seed Points, or**
  - Mix uniformly distributes nodes and nodes generated with seed points.

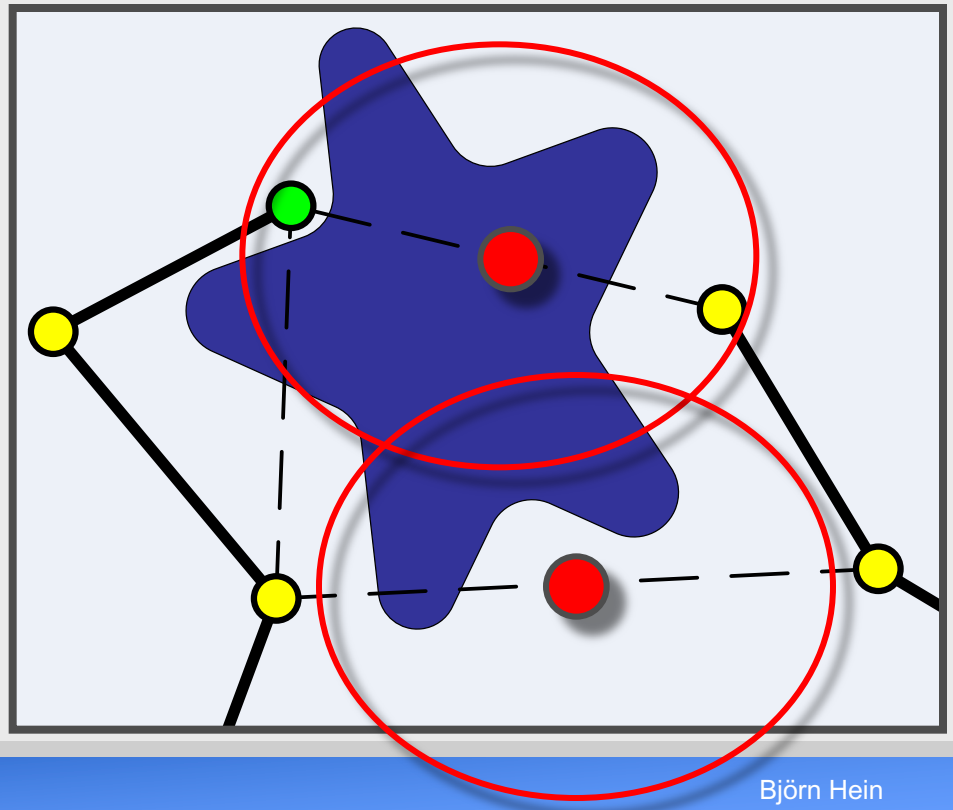
Lazy-PRM: Node enhancement

- If we don‘t create new nodes algorithm fails.
- Strategies:
  - Create again uniformly distributed nodes, or
  - Choose special regions to create nodes: **Seed Points, or**
  - Mix uniformly distributes nodes and nodes generated with seed points.
- Possible way of generating seed points
  - Seed points in the middle of colliding nodes



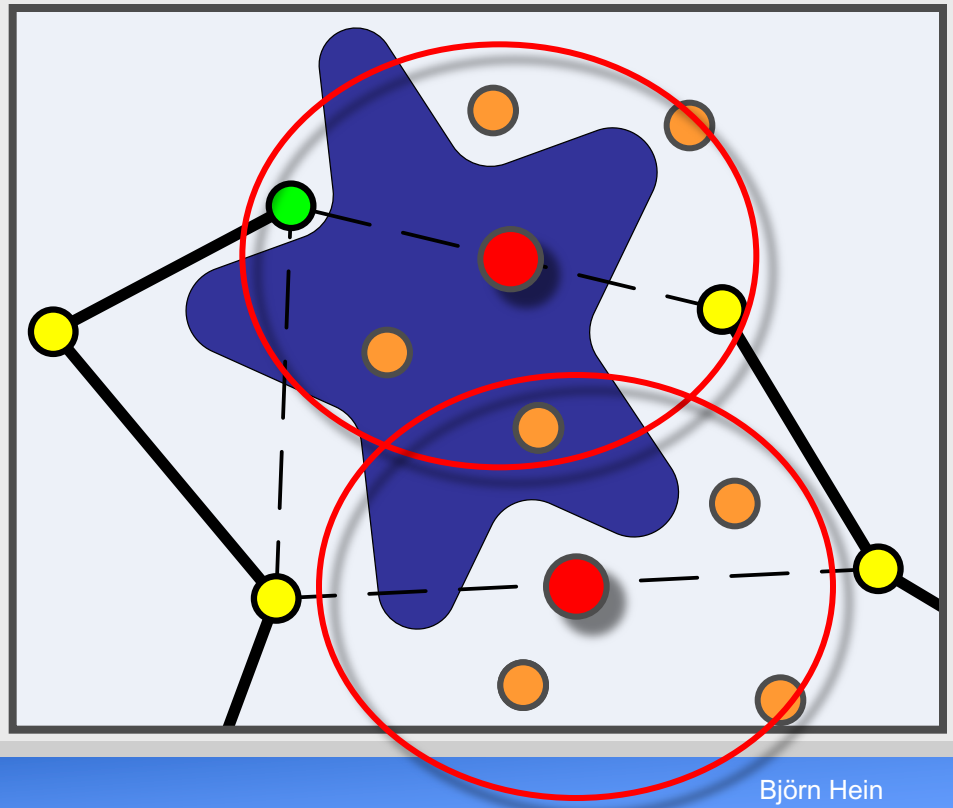
Lazy-PRM: Node enhancement

- If we don‘t create new nodes algorithm fails.
- Strategies:
  - Create again uniformly distributed nodes, or
  - Choose special regions to create nodes: **Seed Points, or**
  - Mix uniformly distributes nodes and nodes generated with seed points.
- Possible way of generating seed points
  - Seed points in the middle of colliding nodes
  - Generate randomly new nodes in a region around seed points



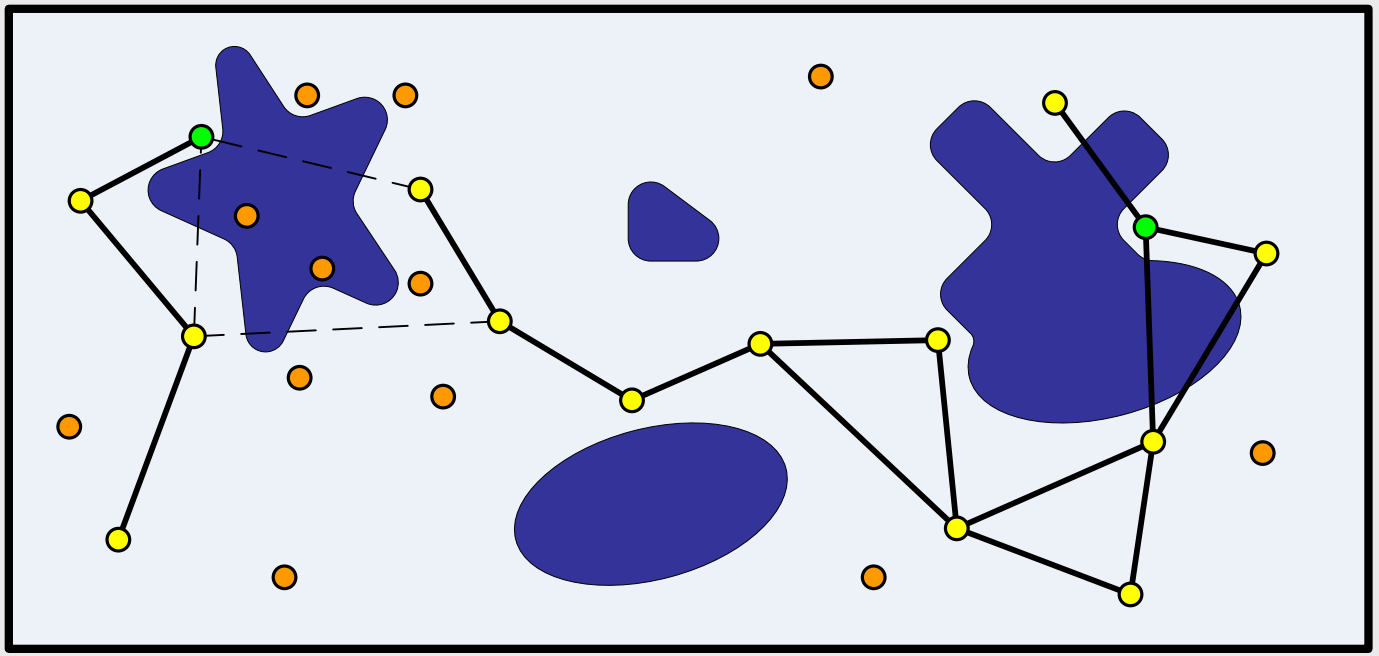
Lazy-PRM: Node enhancement

- If we don‘t create new nodes algorithm fails.
- Strategies:
  - Create again uniformly distributed nodes, or
  - Choose special regions to create nodes: **Seed Points, or**
  - Mix uniformly distributes nodes and nodes generated with seed points.
- Possible way of generating seed points
  - Seed points in the middle of colliding nodes
  - Generate randomly new nodes in a region around seed points



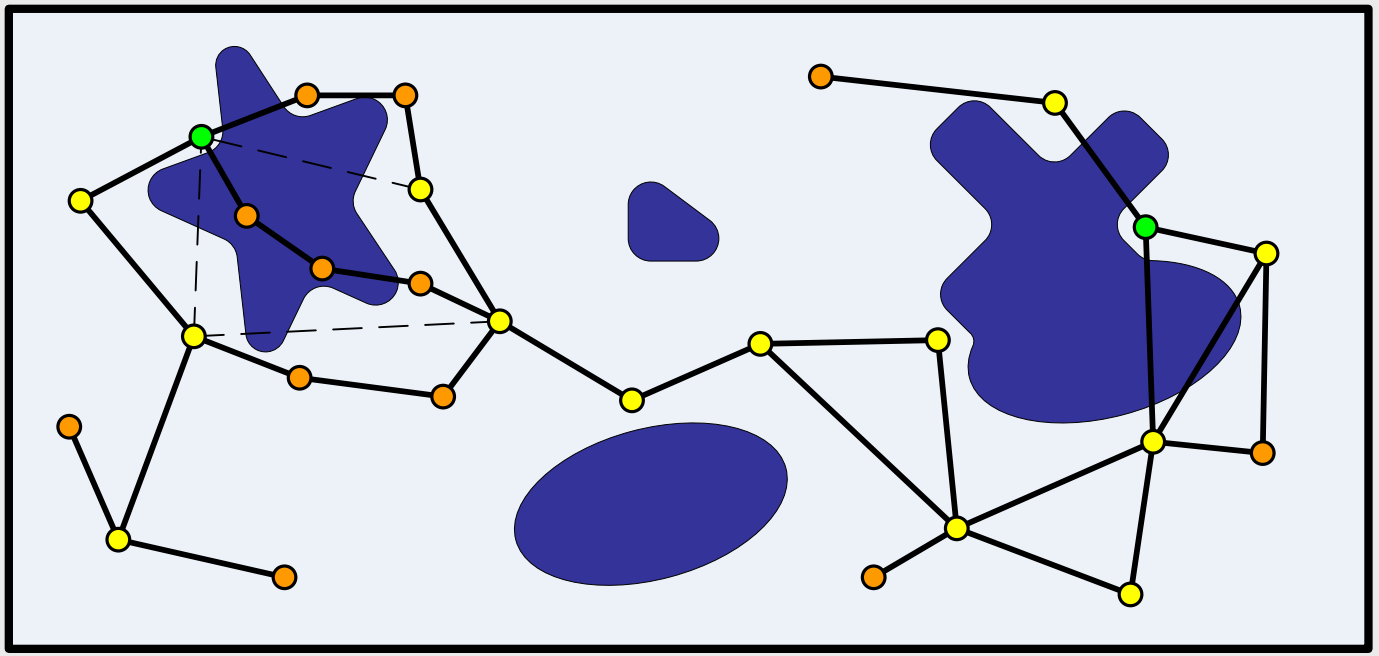
Lazy-PRM: Node enhancement

- Create nodes based on the combination of seed points and uniformly distributed nodes, **don‘t care about collisions**

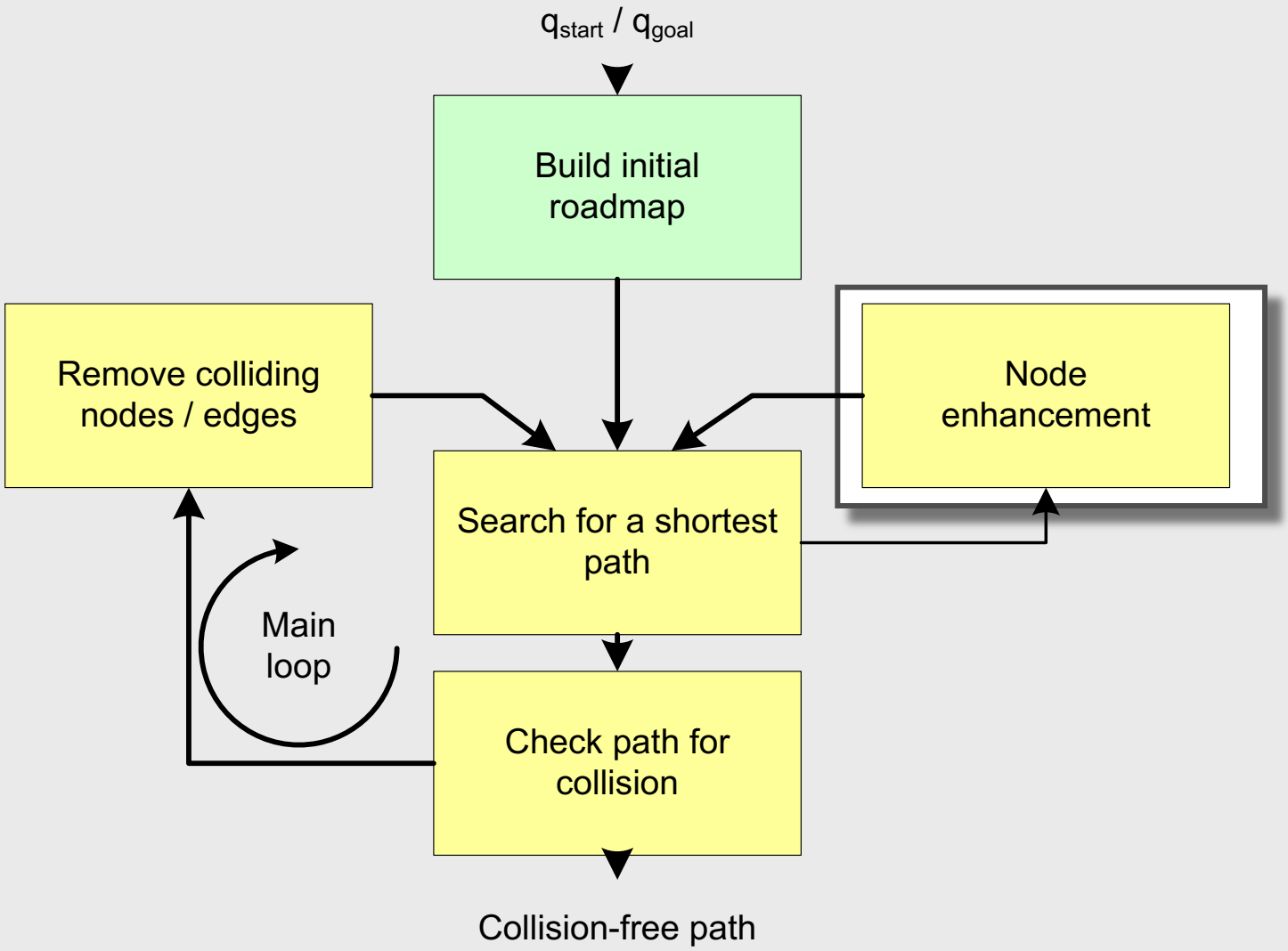


Lazy-PRM: Node enhancement

- Create nodes based on the combination of seed points and uniformly distributed nodes, **don‘t care about collisions**
- Create roadmap (k-closest strategy, in example not all created edges are shown to keep drawing and example simple ☺)

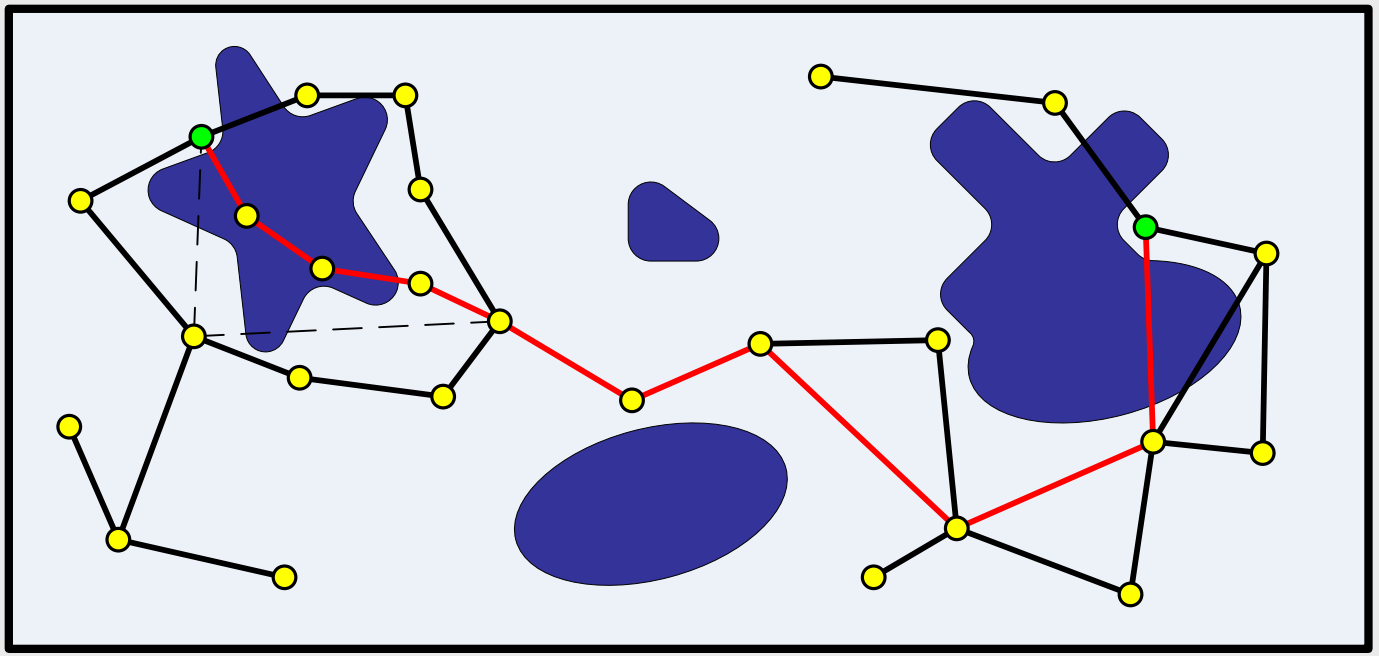


Lazy-PRM: The flow.....



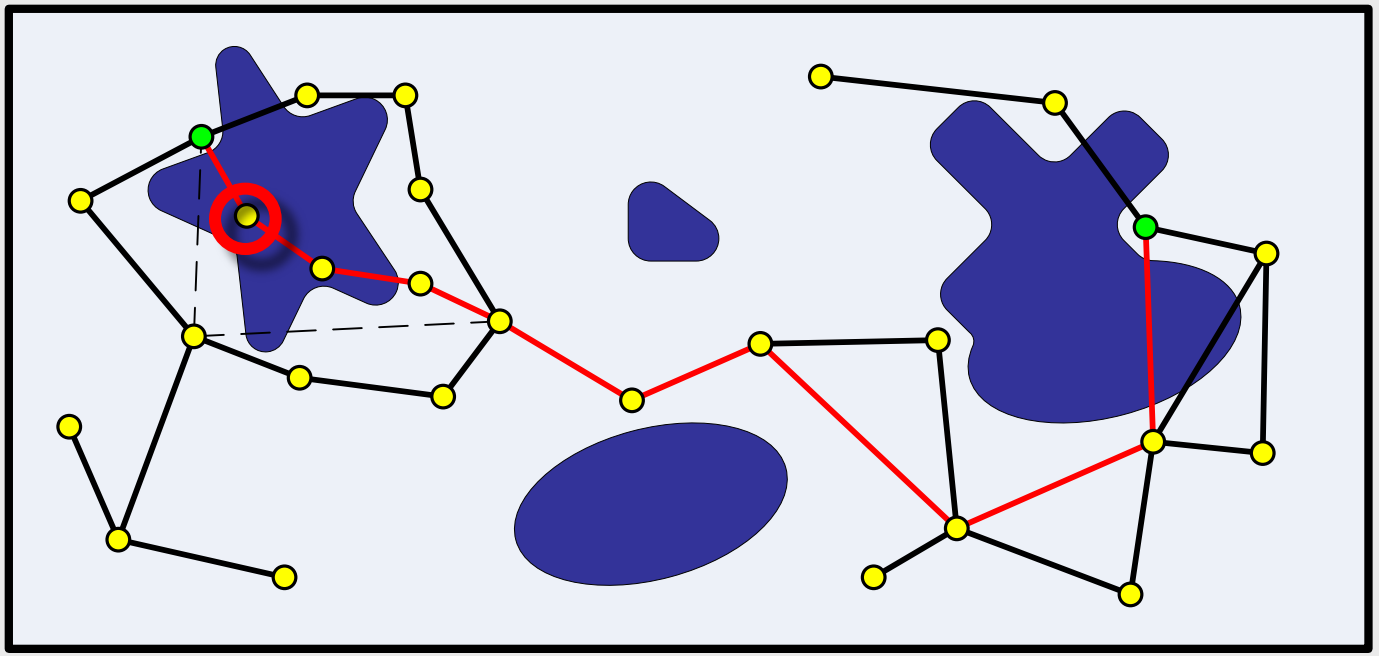
Lazy-PRM: Search again shortest path

- Search shortest path on computed Roadmap, **don‘t care about collisions**.
  - Using A* or Dijkstra



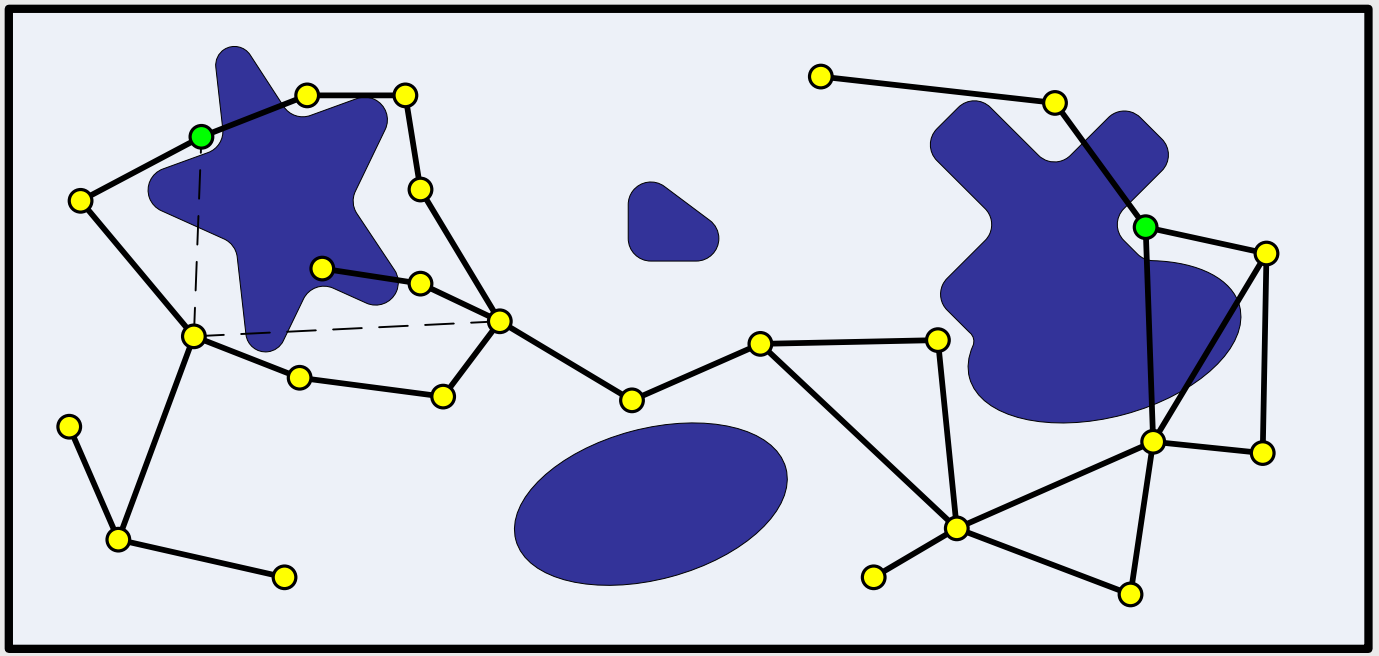
Lazy-PRM: Check nodes /path for collisions

- Check nodes for collisions
  - Remove colliding node



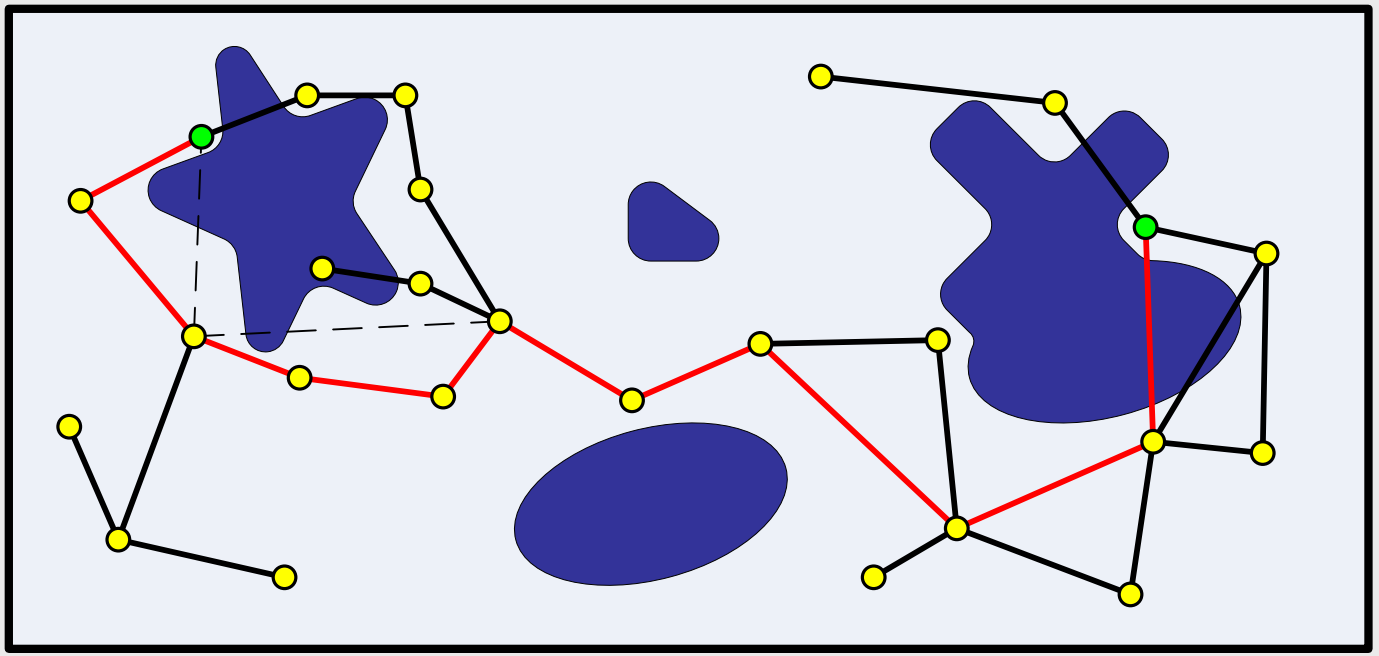
Lazy-PRM: Check nodes /path for collisions

- Check nodes for collisions
  - Remove colliding node



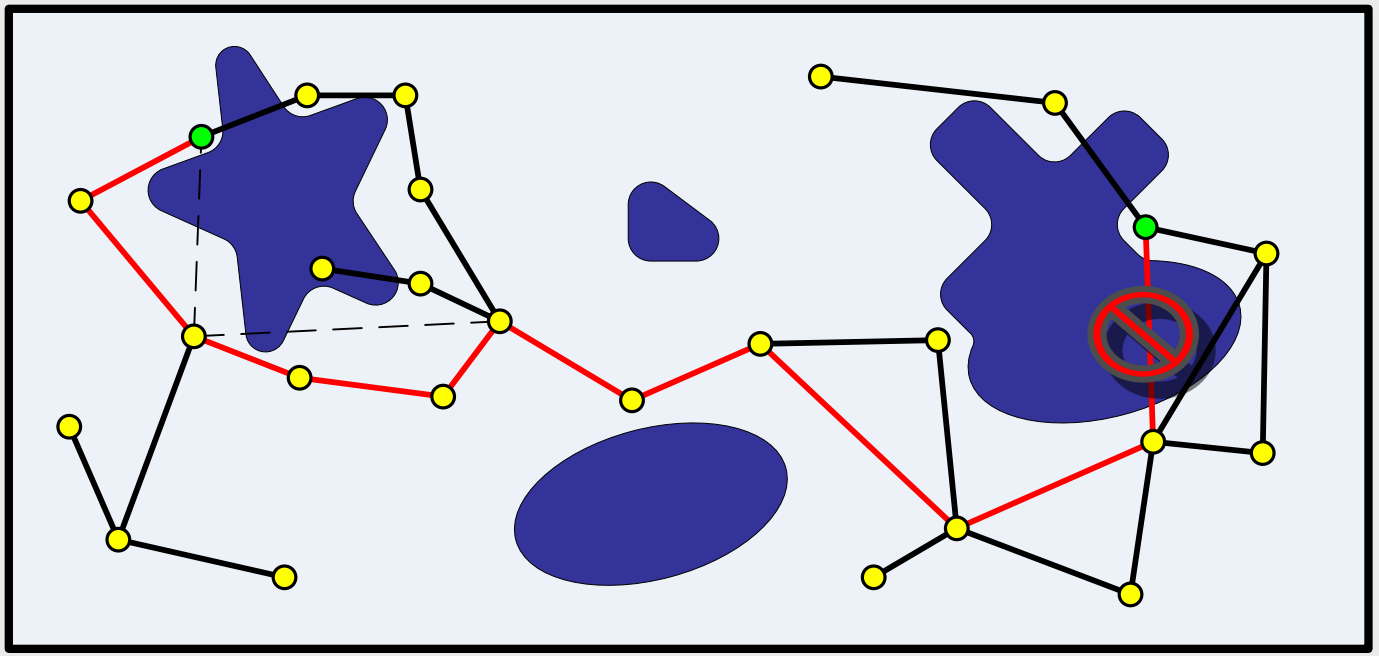
Lazy-PRM: Search again shortest path

- Search shortest path on computed Roadmap, **don‘t care about collisions**.
  - Using A* or Dijkstra



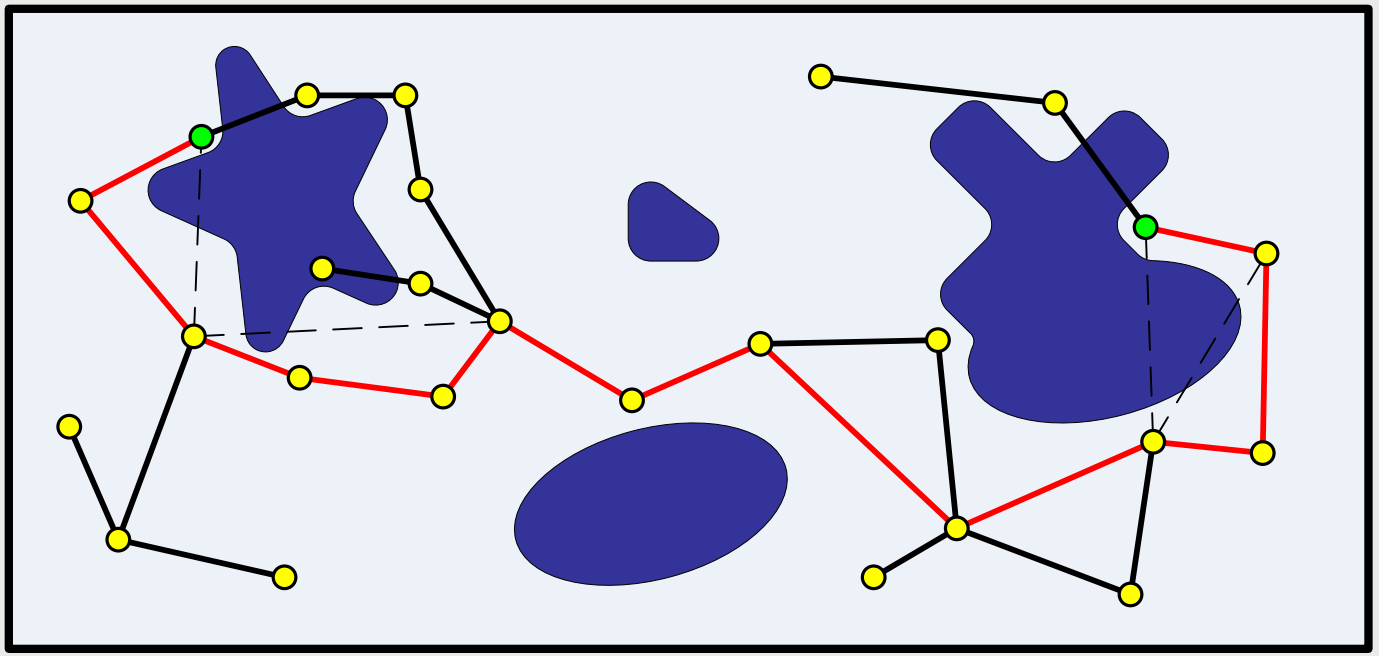
Lazy-PRM: Check nodes /path for collisions

- Check nodes for collisions
- Check edges for collisions



Lazy-PRM

- And so on….. ´til path is found!



Lazy-PRM: in short

1. Create roadmap containing start & goal
2. Randomly generate # configurations without testing them for collision and create corresponding nodes in graph
3. Generate (like K-closest-PRM) for every node edges with k-closest neighbors
4. Search a path with A* connecting start and goal
5. Test alternately from start and goal whether **nodes** of found path are collision free
   - **Stop** if collision is detected and **remove colliding node**
   - **Stop** if alternating search from start and search from goal meets in the middle
6. Test alternately from start and goal whether **edges** of found path are collision free
   - Use local path planner
     - If all edges are collision free → path is found
     - If collision mark edge as colliding and as long as start and goal are connected go back to 4. otherwise go to 7.
7. Generate new points (use some strategy) go to 3.

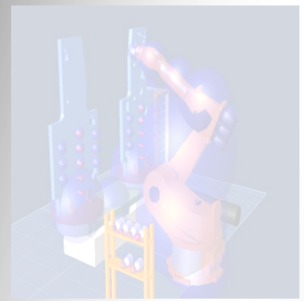
Lazy-PRM: Generating new nodes

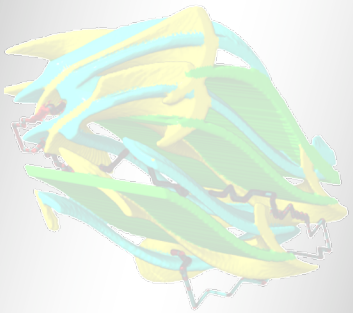
- If start and goal are not connected anymore due to removal of colliding nodes or edges, new nodes have to be generated.
- Possible strategies:
  - Randomly generate nodes (like in 1.) overall in C-space
  - **More sophisticated way**: using information about colliding regions (**seed points & expansion)**
    - Regions give information about existing obstacles
    - New points nearby obstacles can open up new collision free paths around these obstacles
    - Two steps are to be done
      1. Chosing seed points
      2. Generating new nodes around seed points (**seed expansion**)

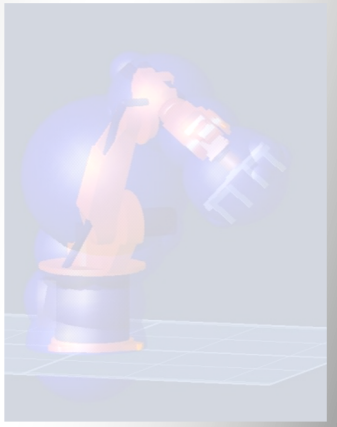
Lazy-PRM: Generating nodes during seed expansion

- Random based expansion
  - Choose a distribution (e.g. gaussian, normal, …) and derivation:
  - Other dependencies (e.g on corresponding link of robot → main axes other derivation than hand axes)
- Distance based expansion
  - Generate in free C-space around seed points new nodes
  - Use free distances as minimal step size for distance of new nodes

Lazy-Variants: SBL







Single-query

Bi-directional

Lazy-collision-checking

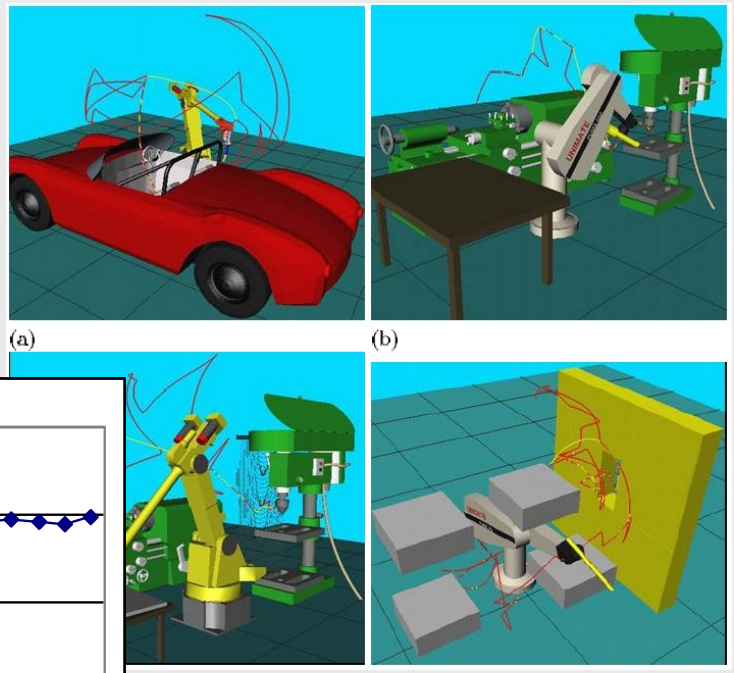
(G. Sánchez and J. Latombe)

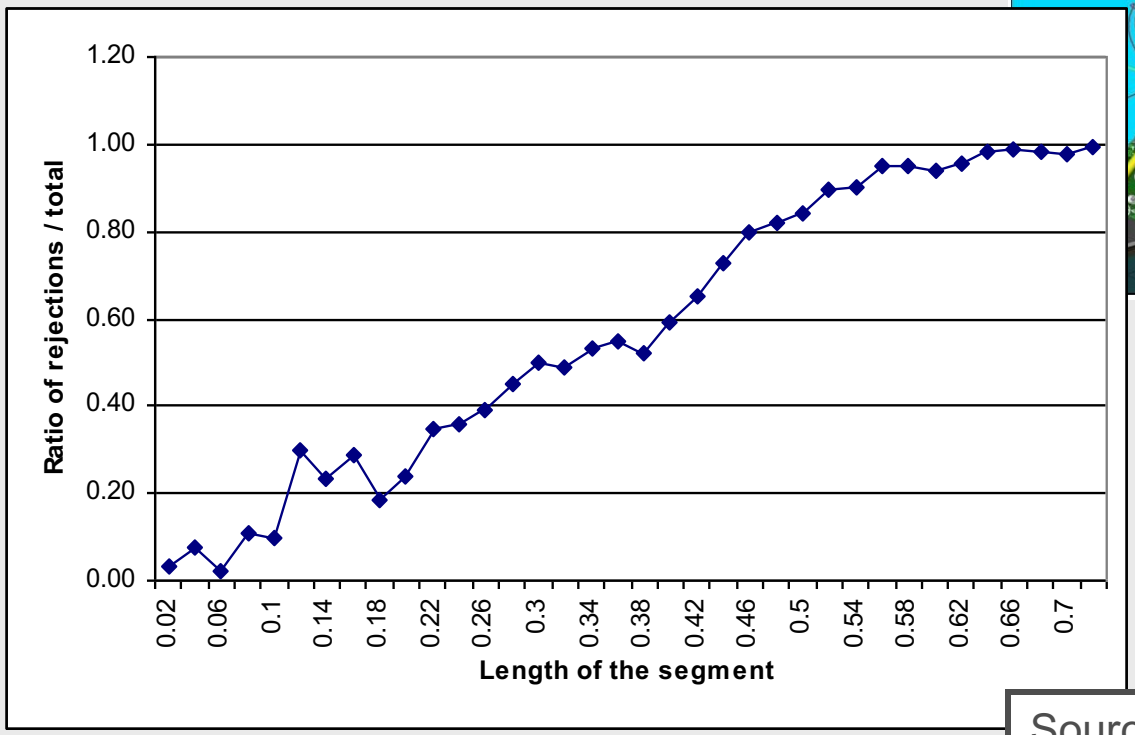
Some issues still open

- Potential of „lazy collision-checking“ only partially exploited
- In advance to decide how large network should be
  - To coarse: it‘ll fail
  - To dense: wasting time checking similar paths
- Focus on shortest paths could be to expensive regarding planning time → it‘s better to focus on finding fast a collision-free path

Probability of collision depending on segment length

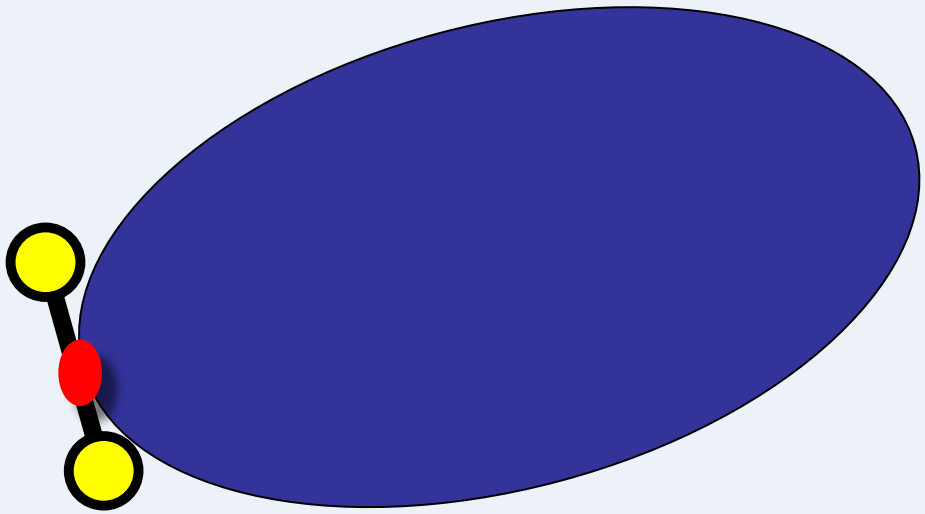
- 100 randomly generated **free** configurations
- Connect these configurations to 100 other randomly generated ones
- Getting 10000 segments





Source: presentation by Niloy J. Mitra

Why are there few short colliding segements?



A short **colliding** segment with **collision free endpoints** is necessarily almost tangential to an C-Space obstacle → small probability of happening

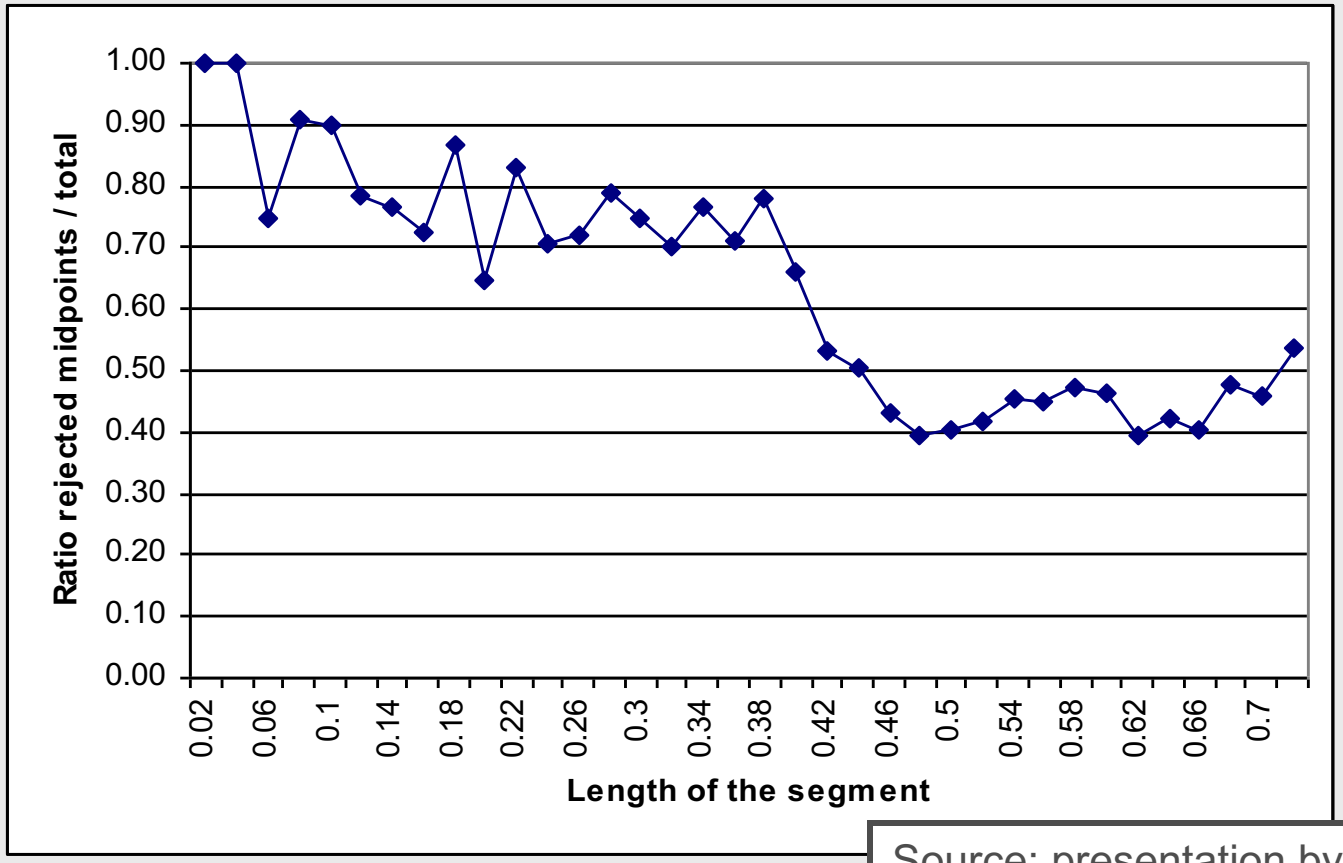
Background: Experimental foundations

Observations when using PRMs:

- Most local paths are not on the final path
- Collision-free tests are most expensive
- Short connections between two milestones have high prior probabilities of being free
- If a connection is colliding, it's midpoint has high probability of being in collision
  - Next slide…

If a connection is colliding - probability of colliding Midpoint is high

- Adapt therefore line-testing strategy



Source: presentation by Niloy J. Mitra

SBL

- Combining three aspects:
  - Single-query
  - Bi-directional
  - Adaptiv sampling strategies
- Incrementally construct network from **two trees** starting at start & goal
- Focusing search at regions beeing reachable from these trees
- Adapt sampling resolution depending on free space
  - Big steps in open regions
  - small steps in narrow passages
- Connections are not immediately tested for collision
- Only when a sequence of nodes are joining start and goal the edges are tested for collision

SBL: Overall algorithm

Input:

s - maximal number of milestones that it is allowed to generate

ρ **-** distance threshold. Two configurations are considered *close* if their distance is less then ρ.

1. Insert q_{start} and q_{goal} as the roots of T_{start} and T_{goal} respectively
2. Repeat s times
   a. EXPAND-TREE
   b. τ ← CONNECT-TREE
   c. If τ ≠ *nil* then return τ
3. Return *failure*

SBL: Overall algorithm

Input:

s - maximal number of milestones that it is allowed to generate

ρ **-** distance threshold. Two configurations are conidered *close* if their distance is less then ρ.

1. Insert q_{start} and q_{goal} as the roots of T_{start} and T_{goal} respectively
2. Repeat s times
   a. EXPAND-TREE
   b. τ ← CONNECT-TREE
   c. If τ ≠ *nil* then return τ
3. Return *failure*

= Add a node to one of the two trees

SBL: Expand Tree

B(q, r) = { q' ∈ C | d(q,q') < r }: neighborhood of q of radius r

π(v) ~ 1/η(v); η(v) = Density of nodes around v

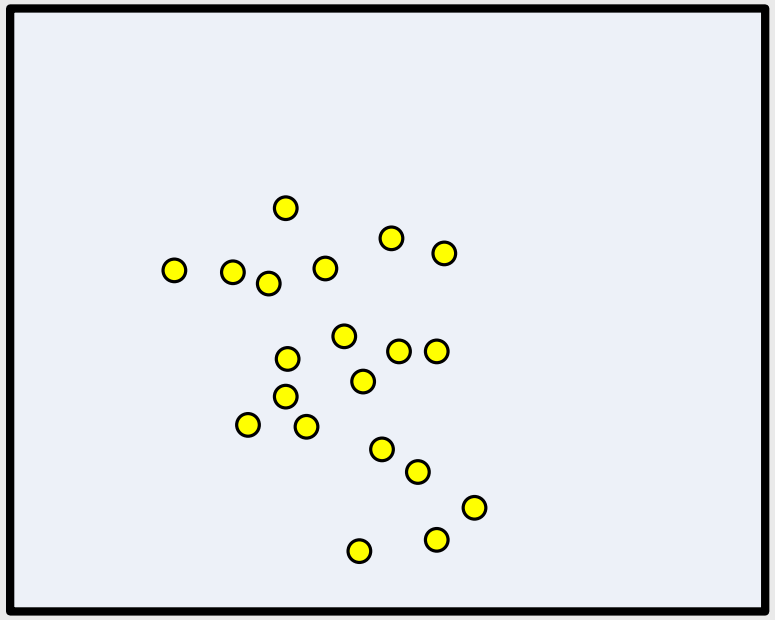
1. Pick T to be either T_{start} or T_{goal}, each with P=0.5
2. Repeat until a node is generated
   1. Pick a node v at random, with Probability $\pi(v)$
   2. For i = 1,2,… until a new q been generated
      1. configuration q uniformly at random from $B(v, \rho/i)$
      2. If q is collision-free, then install it as a child of v in T
         (Note: collision-test of segment is not done here)

How to compute „density“ in a fast way

- Create to arrays A_{start} and A_{goal} (DIM = 2 or 3)
- Both arrays partition space in equally sized cells
- When a new node is added to a tree T_{start} or T_{goal}, corresponding A_{start}/A_{goal} is updated.
- Chose node depending on density:
  - Pick a non-empty cell of A_{xxx} with probability depending on number of nodes in cell
  - Pick a node of this cell

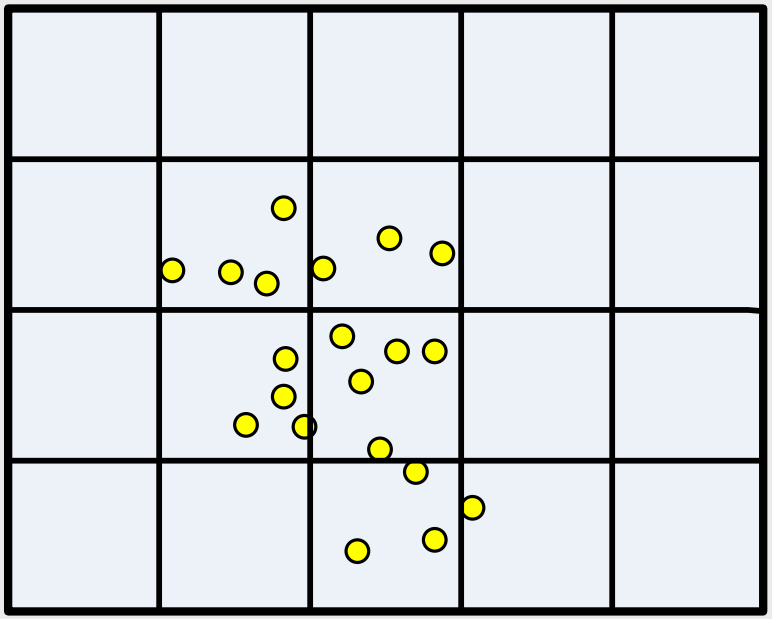
Idea behind cell-density

- Don‘t maintain density around each node → overhead



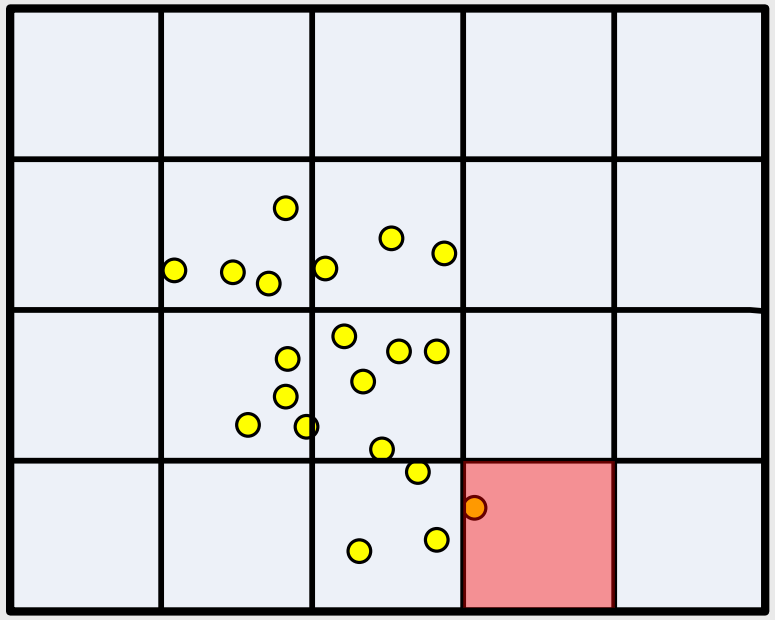
Idea behind cell-density

- Don‘t maintain density around each node → overhead
- Use equaly seized cells and maintain density in these cells



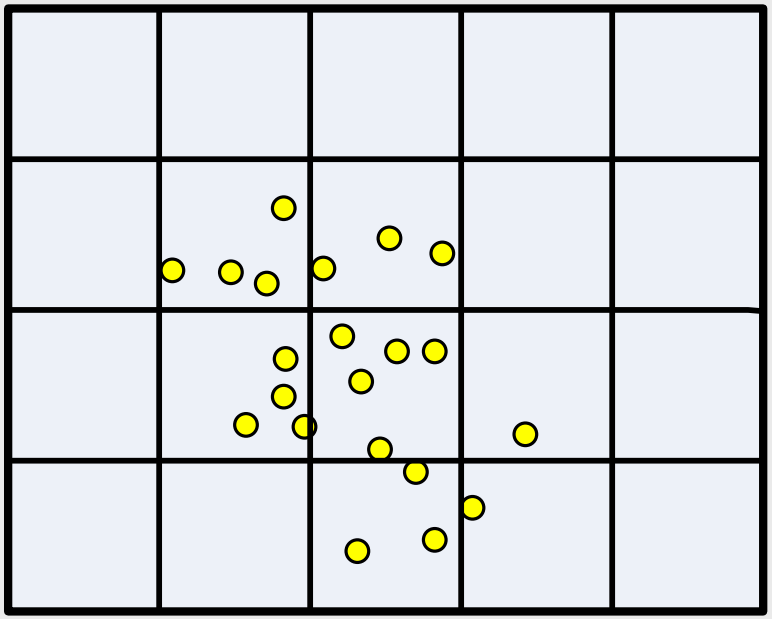
Idea behind cell-density

- Don‘t maintain density around each node → overhead
- Use equaly seized cells and maintain density in these cells
- Choose cell having lowest density whith highest probability



Idea behind cell-density

- Don‘t maintain density around each node → overhead
- Use equaly seized cells and maintain density in these cells
- Choose cell having lowest density whith highest probability
- Generate new node



SBL: Overall algorithm

Input:

s - maximal number of milestones that it is allowed to generate

ρ **-** distance threshold. Two configurations are conidered *close* if their distance is less then ρ.

1. Insert q_{start} and q_{goal} as the roots of T_{start} and T_{goal} respectively
2. Repeat s times
   a. EXPAND-TREE
   b. τ ← CONNECT-TREE
   c. If τ ≠ *nil* then return τ
3. Return *failure*

= Connect the two trees

SBL: Connect Tree

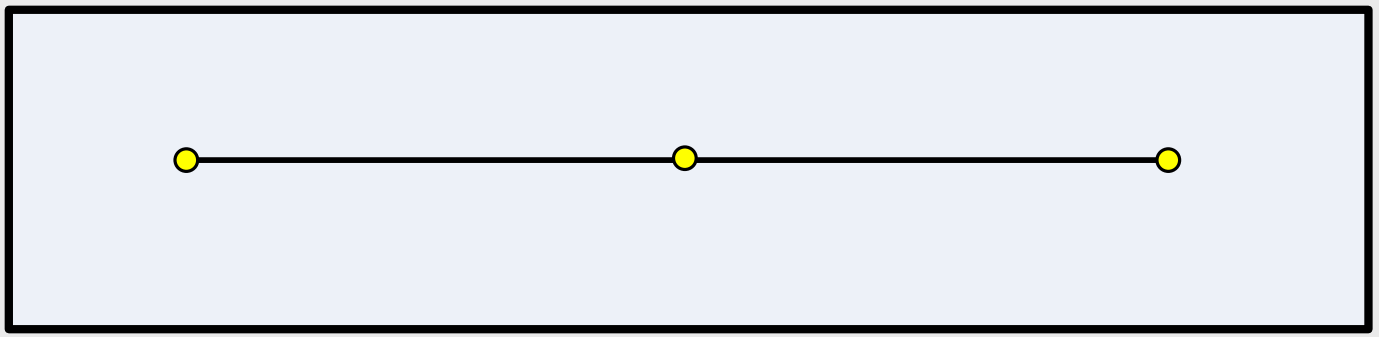
1. $v \leftarrow$ most recently created node
2. $v' \leftarrow$ closest node to v in the other tree
3. If $d(v,v') < \rho$ then
    1. Connect v and v' by a bridge w
    2. $\tau \leftarrow$ path connecting q_{start} and q_{goal}
    3. Return TEST-PATH
4. Return *nil*

SBL: Path testing

- SBL uses collision-check index $\kappa(u)$ rating the amount of testing done on an edge u
- $\kappa(u) = 0 \rightarrow$ only two endpoints of u are tested

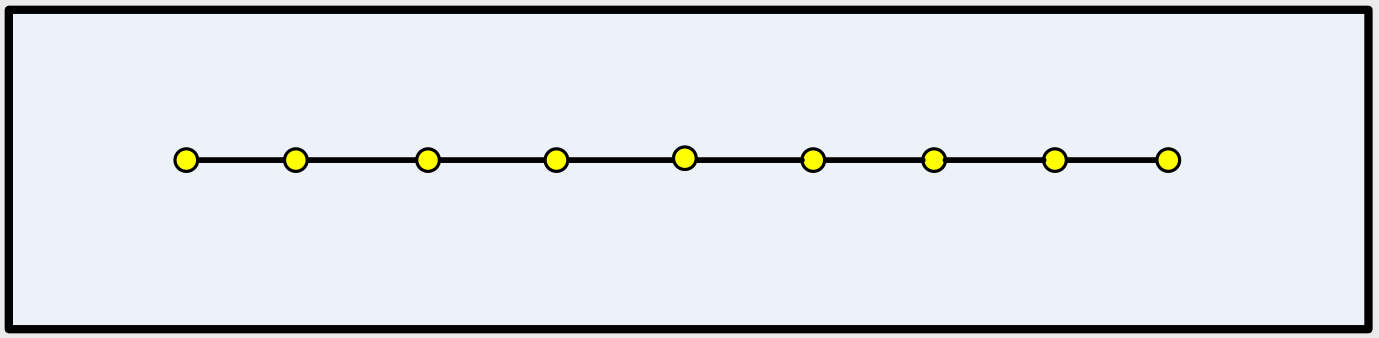
SBL: Path testing

- SBL uses collision-check index $\kappa(u)$ rating the amount of testing done on an edge u
- $\kappa(u) = 0$ → only two endpoints of u are tested
- $\kappa(u) = 1$ → two endpoints and midpoint are tested



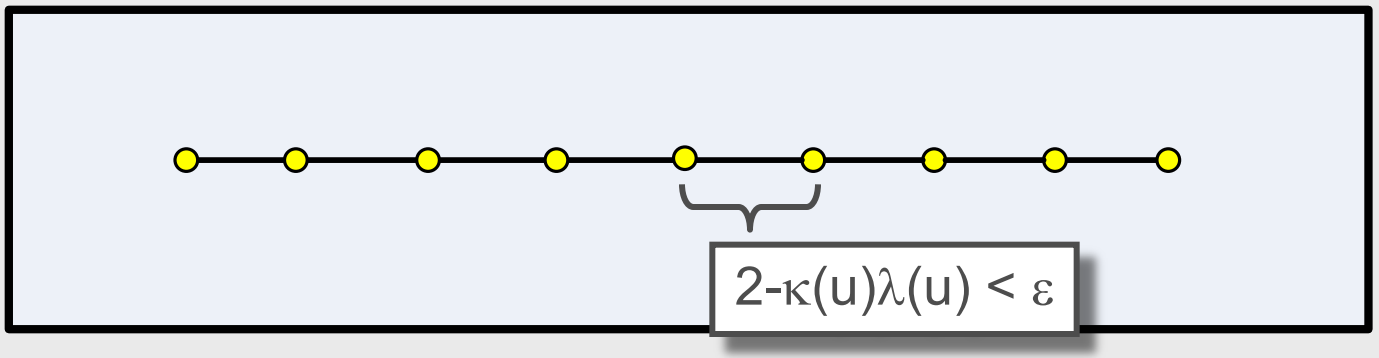
SBL: Path testing

- SBL uses collision-check index $\kappa(u)$ rating the amount of testing done on an edge u
- $\kappa(u) = 0$ → only two endpoints of u are tested
- $\kappa(u) = 1$ → two endpoints and midpoint are tested
- For a given $\kappa(u)$, $2^{\kappa(u)}+1$ equally distant points are tested and are collision-free



SBL: Path testing

- SBL uses collision-check index $\kappa(u)$ rating the amount of testing done on an edge u
- $\kappa(u) = 0 \rightarrow$ only two endpoints of u are tested
- $\kappa(u) = 1 \rightarrow$ two endpoints and midpoint are tested
- For a given $\kappa(u)$, $2^{\kappa(u)}+1$ equally distant points are tested and are collision-free
- If $\lambda(u) = \text{length}(u) \rightarrow$ u is save if $2^{-\kappa(u)}\lambda(u) < \varepsilon$



SBL: Test edge

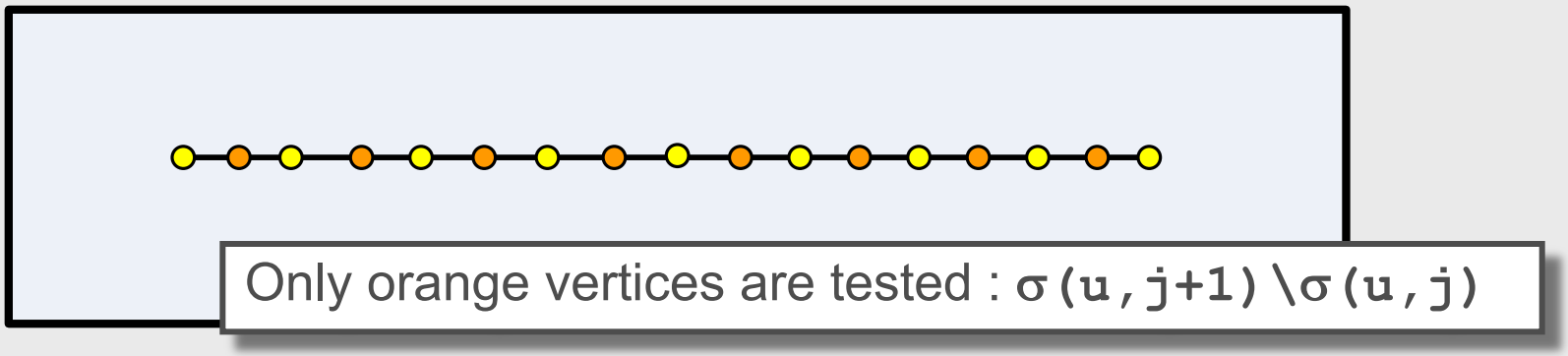
- Let $\sigma(u,j)$ be the set of points along beeing already tested collision-free in order for $\kappa(u)$ to have value j
- Algorithm Test-Segments increase $\kappa(u)$ by one

```
1.  j ← κ(u)
```

SBL: Test edge

- Let $\sigma(u,j)$ be the set of points along beeing already tested collision-free in order for $\kappa(u)$ to have value j
- Algorithm Test-Segments increase $\kappa(u)$ by one

1. `j ← κ(u)`
2. `for every q ∈ σ(u,j+1)\σ(u,j) if q is in collision return collision`

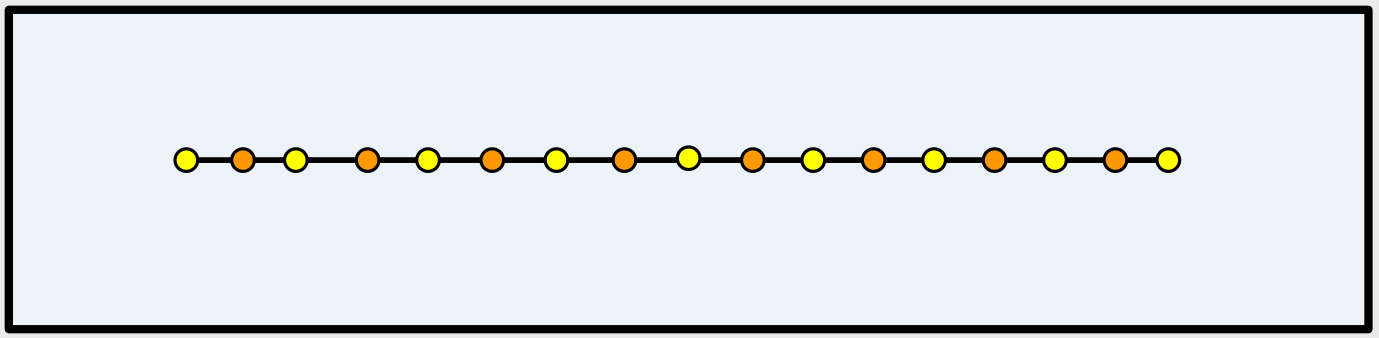


SBL: Test edge

- Let $\sigma(u,j)$ be the set of points along beeing already tested collision-free in order for $\kappa(u)$ to have value j
- Algorithm Test-Segments increase $\kappa(u)$ by one

1. $j \leftarrow \kappa(u)$
2. **for every** $q \in \sigma(u,j+1) \setminus \sigma(u,j)$ **if** q **is in collision return** *collision*
3. **if** $2^{-(j+1)} \lambda(u) < \varepsilon$ **mark** u **as safe else** $\kappa(u) \leftarrow j + 1$

For every edge not beeing safe, the value $2^{-\kappa(u)}\lambda(u)$ is cached.



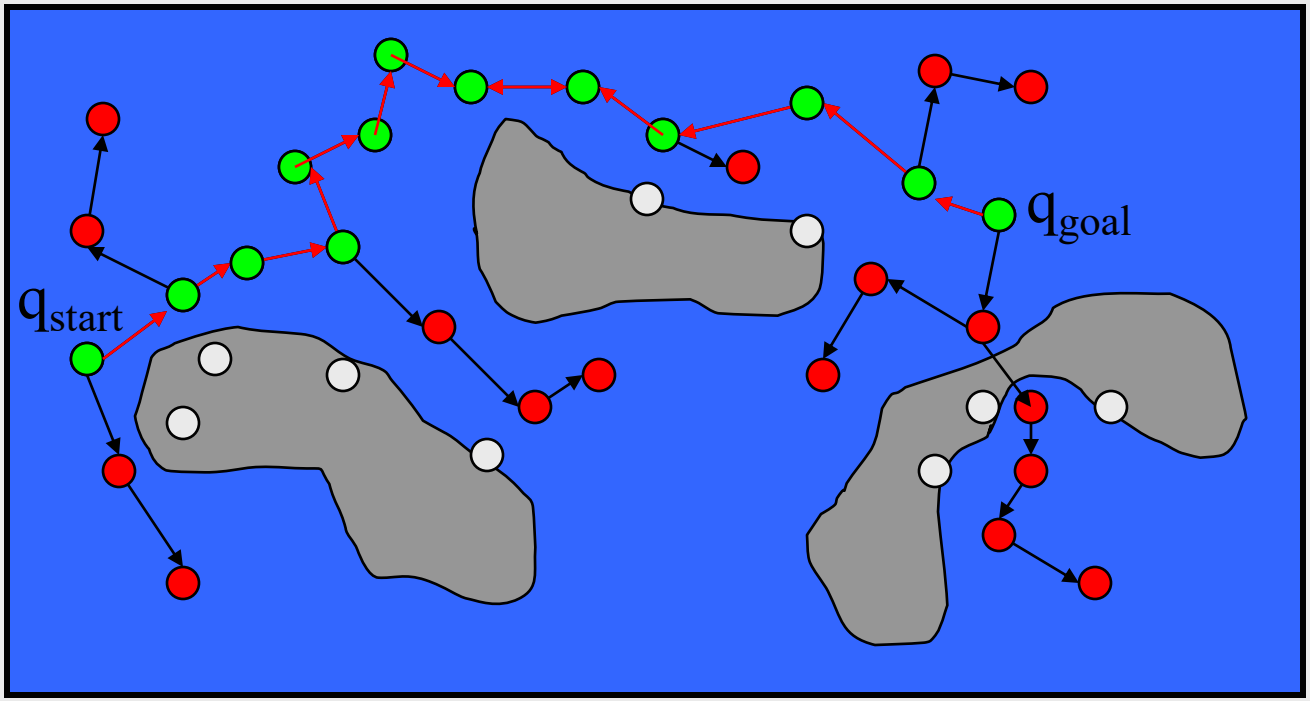
SBL: Test Path

- $u_1, u_2, \ldots, u_p$ denote all edges in path that are not marked save
- U is a priority queue of these edges sorting by decreasing order of $2^{-\kappa(u_i)}\lambda(u_i)$ (i = 1 to p).

```
TEST-PATH(τ)
PriorityQueue U;
1. While U is not empty do
    1. u ← extract(U)
    2. If TEST-SEGMENT(u) = collision then
        1. Remove u from the roadmap
        2. Return nil
    3. If u is not marked safe then re-insert u into U
2. Return τ
```

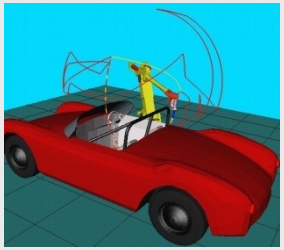
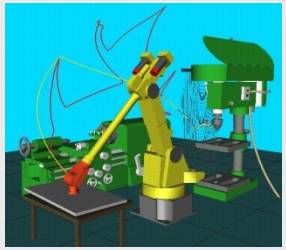
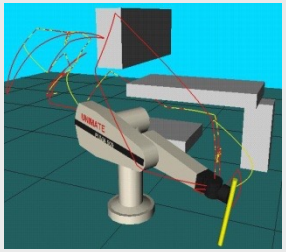
SBL: Example

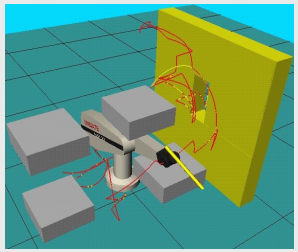
- Example search
  - Two search trees are



Source: presentation by Niloy J. Mitra

Performance – BOOST……



Average performance with lazy collision checking

Example	Running Time(secs)	Milestones in Roadmap	Milestones in Path	Total Nr of Collision Checks	Collision Checks on the Path	Sampled Milestones	Comput. Time for Coll-Check (secs)	Std. Deviation for running time
1a	0.60	159	13	1483	342	245	0.58	0.38
1b	4.45	1609	39	11211	411	7832	4.21	2.48
1c	4.42	1405	24	7267	277	3769	4.17	1.86
1d	0.17	33	10	406	124	47	0.17	0.07
1e	6.99	4160	44	12228	447	6990	6.30	3.55

Average performance without lazy collision checking

Example	Running Time(secs)	Milestones in Roadmap	Milestones in Path	Total Nr of Collision Checks	Collision Checks on the Path	Sampled Milestones	Comput. Time for Coll-Check (secs)	Std. Deviation for running time
1a	2.82	22	5	7425	173	83	2.81	3.01
1b	106.20	3388	32	300060	421	9504	105.56	59.30
1c	18.46	771	16	38975	219	3793	18.35	15.34
1d	1.03	29	9	2440	123	46	1.02	0.70
1e	293.77	6737	24	666084	300	11971	292.40	122.75

Source: presentation by Niloy J. Mitra