Using the GPU for motion planning

Algorithms and Data Structures 2 – Motion Planning and its applications

University of Applied Sciences Stuttgart

Dr. Daniel Schneider

Introduction

- Nowadays GPUs has gained a lot in relevance for computations.
- Applications like AI, Data Mining, Blockchain, Video games are heavily using the GPU.
- Nowadays even smartphone include GPUs with substantial shader cores. Robotics applications start to use these for online planning.
- In the past effort were made to bring motion planners to GPUs.
- There are two major papers for parallelizing motion planners:
  - 1 paper for RRT
  - 1 paper for PRM

For PRM – g-Planner

- PRM algorithms are by design easy to parallelized.
- Nevertheless there are multiple questions that arise during implementation.
- The implementation of this algorithm is also available as

g-Planner: Real-time Motion Planning and Global Navigation using GPUs

Jia Pan and **Christian Lauterbach** and **Dinesh Manocha**
Department of Computer Science, University of North Carolina at Chapel Hill
{panj, cl, dm}@cs.unc.edu
http://gamma.cs.unc.edu/gplanner/

Abstract

We present novel randomized algorithms for solving global motion planning problems that exploit the computational capabilities of many-core GPUs. Our approach uses thread and data parallelism to achieve high performance for all components of sample-based algorithms, including random sampling, nearest neighbor computation, local planning, collision queries and graph search. This approach can efficiently solve both the multi-query and single-query versions of the problem and obtain considerable speedups over prior CPU-based algorithms. We demonstrate the efficiency of our algorithms by applying them to a number of 6DOF planning benchmarks in 3D environments. Overall, this is the first algorithm that can perform real-time motion planning and global navigation using commodity hardware.

Introduction

Motion planning is one of the fundamental problems in algorithmic robotics. The classical formulation of the problem is: given an arbitrary robot, R, and an environment composed of obstacles, compute a continuous collision-free path for R from an initial configuration to the final configuration. It is also known as the *navigation problem* or *piano mover's problem*. Besides robotics, motion planning algorithms are also used in CAD/CAM, computer animation, computer gaming, computational drug-design, manufacturing, medical simulations etc.

There is extensive literature on motion planning and global navigation. At a broad level, they can be classified into local and global approaches. The local approaches, such as those based on artificial potential field methods (Khatib 1986), are quite fast but not guaranteed to find a path. On the other hand, global methods based on criticality analysis or roadmap computation (Schwartz and Sharir 1983; Canny 1988) are guaranteed to find a path. However, the complexity of these exact or complete algorithms increases as an exponential function of the number of degrees-of-freedom (DOF) of the robot and their implementations have been restricted to only low DOF.

Practical methods for global motion planning for high-DOF robots are based on randomized sampling (Kavraki et al. 1996; LaValle and Kuffner 2000). These methods attempt to capture the topology of the free space of the robot by generating random configurations and connect nearby configurations using local planning methods. The resulting algorithms are probabilistically complete and have been successfully used to solve many high-DOF motion planning and navigation problems in different applications. However, they are too slow for interactive applications or dynamic environments.

Main Results: We present a novel parallel algorithm for real-time motion planning of high DOF robots that exploits the computational capability of a $400 commodity graphics processing unit (GPU). Current GPUs are programmable many-core processors that can support thousands of concurrent threads and we use them for real-time computation of a probabilistic roadmap (PRM) and a lazy planner. We describe efficient parallel strategies for the construction phase that include sample generation, collision detection, connecting nearby samples and local planning. The query phase is also performed in parallel based on graph search. In order to design an efficient single query planner, we use a lazy strategy that defers collision checking and local planning. In order to accelerate the overall performance, we also describe new hierarchy-based collision detection algorithms.

The performance of the algorithm is governed by the topology of the underlying free space as well as the methods used for sample generation and nearest neighbor computation. In practice, our algorithm can generate thousands of samples for robots with 3 or 6 DOFs and compute the roadmap for these samples at close to interactive rates including construction of all hierarchies. It performs no precomputation and is applicable to dynamic scenes, articulated models or non-rigid robots. We highlight its performance on multiple benchmarks on a commodity PC with a NVIDIA GTX 285 GPU and observe a 10-80 times performance improvement over CPU-based implementations.

The rest of the paper is organized as follows. We survey related work on motion planning and GPU-based algorithms in Section 2. Section 3 gives an overview of our approach and we present parallel algorithms for the construction and query phase in Section 4. We highlight our performance on different motion planning benchmarks in Section 5 and compare with prior methods.

Sources:
g-Planner: Real-time Motion Planning and Global Navigation using GPUs – Jia Pan and Christian Lauterbach and Dinesh Manocha – http://gamma.cs.unc.edu/gplanner/AAAI.pdf

The Approach

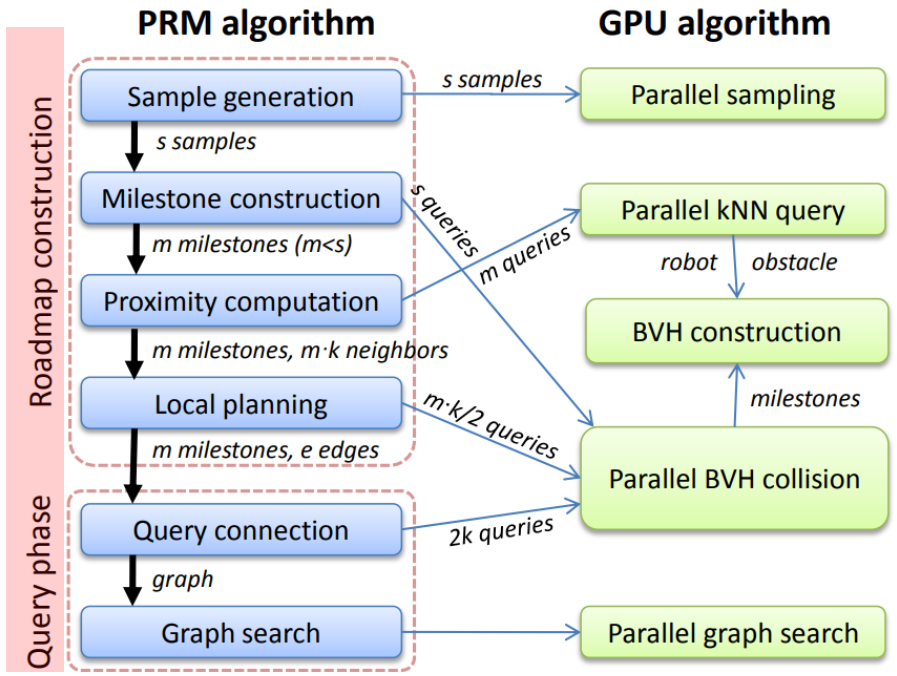


Figure 2: PRM overview and parallel components in our algorithm

Sources:
g-Planner: Real-time Motion Planning and Global Navigation using GPUs – Jia Pan and Christian Lauterbach and Dinesh Manocha – http://gamma.cs.unc.edu/gplanner/AAAI.pdf

- On the left-hand side you see the standard PRM motion planner.
- More precisely, the sPRM.
- On the right-hand side you see the implementation and data structures how the GPU is utilized in with an GPU-based sPRM.
- We will now go through all ingredients of this algorithm.

The Approach - Sampling

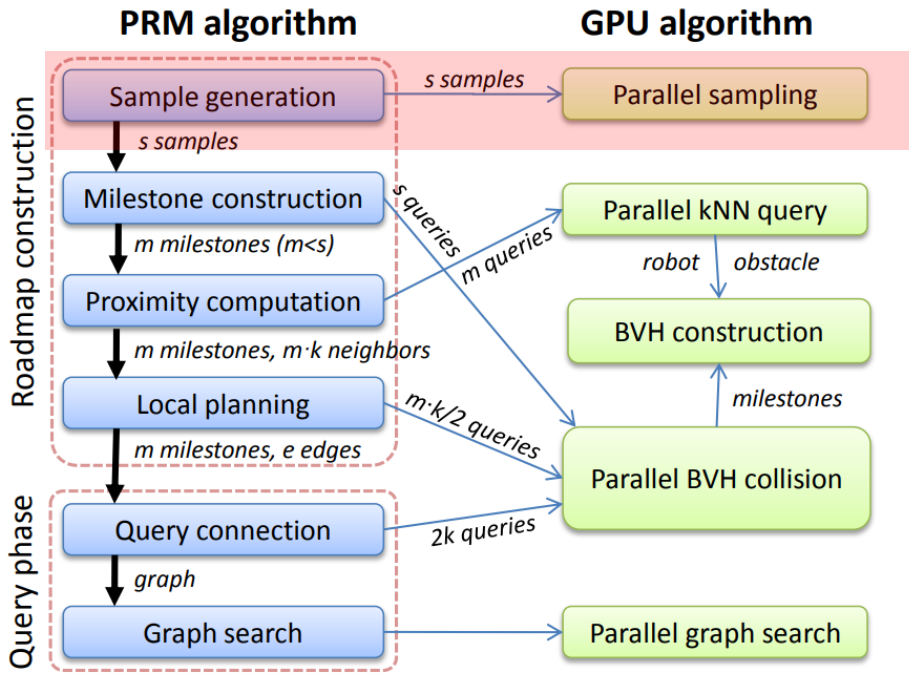


Figure 2: PRM overview and parallel components in our algorithm

Sources:
g-Planner: Real-time Motion Planning and Global Navigation using GPUs – Jia Pan and Christian Lauterbach and Dinesh Manocha – http://gamma.cs.unc.edu/gplanner/AAAI.pdf

- In this phase the algorithm has to compute s amount of random samples in parallel (s threads needed).
- The challenge is that to ensure "real randomness" the same seed has to be used by each thread.
- If every thread has to access the seed, the seed becomes the bottleneck of the implementation and slows down the computation.
- Therefore one has to use a random approach that does not rely on a shared.
- In cryptographic applications there are some examples of these algorithms.

The Approach - Milestone

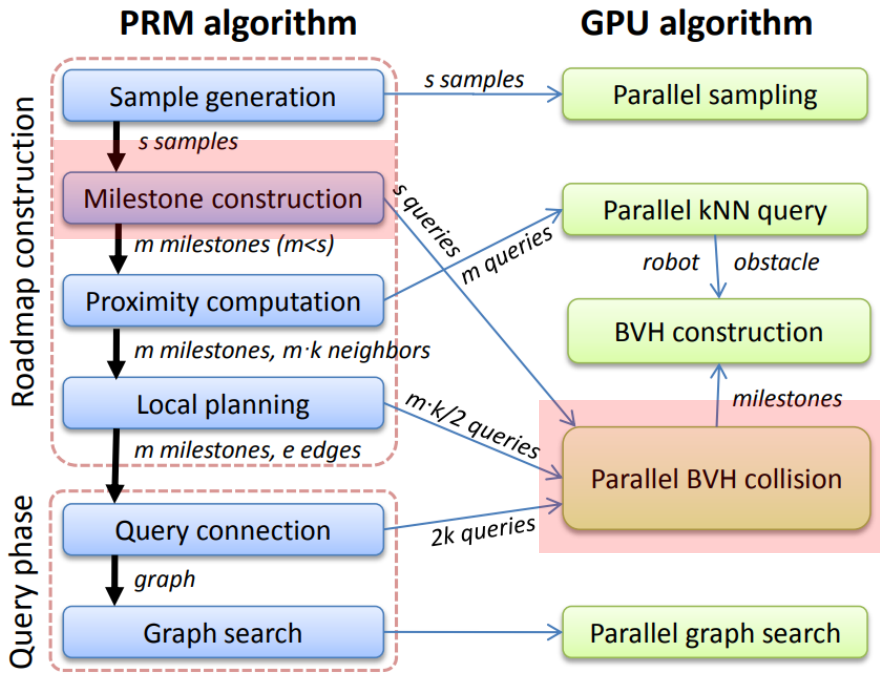


Figure 2: PRM overview and parallel components in our algorithm

Sources:
g-Planner: Real-time Motion Planning and Global Navigation using GPUs – Jia Pan and Christian Lauterbach and Dinesh Manocha – http://gamma.cs.unc.edu/gplanner/AAAI.pdf

- In this paper a milestone is a sampled configuration that is free of collision.
- The paper uses each thread to compute one collision detection (s threads needed).
- The BVH itself is NOT parallelized. Reason: It is not trivial to parallelize the BVH collision detection. There is work in research that does parallelize BVH collision detection but the scaling is not there but not even close to linear.
- There are collision detection algorithms (like VPS) that can are scaling better on GPU. But as s is already very large → not needed.

The Approach - NN

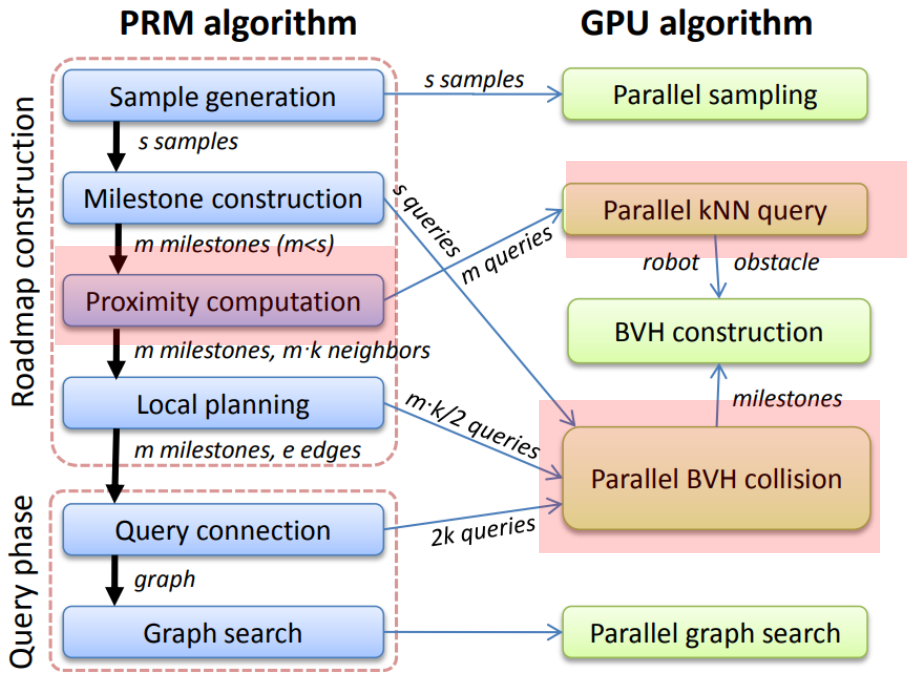


Figure 2: PRM overview and parallel components in our algorithm

Sources:
g-Planner: Real-time Motion Planning and Global Navigation using GPUs – Jia Pan and Christian Lauterbach and Dinesh Manocha – http://gamma.cs.unc.edu/gplanner/AAAI.pdf

- To compute the nearest neighbours the algorithm uses also the BVH.
- For 3 DOFs the BVH can be used to find the exact nearest neighbours.
- For >3 DOFs you have to decompose the spaces and the BVH can only return approximate nearest neighbours.
- Again the BVH is NOT parallelized and only the calls are doe by one thread (m Threads needed)
- A different approach would be to use brute force search or the previously presented LSH approach. With both approaches one could parallelize the query itself as well.
- The paper suggest the LSH.

The Approach – Local planning

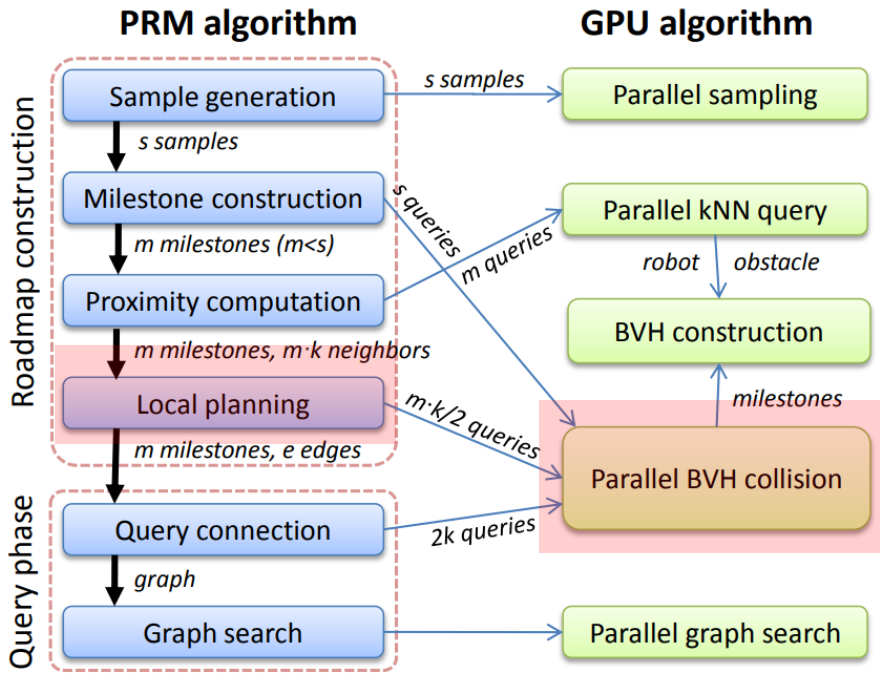
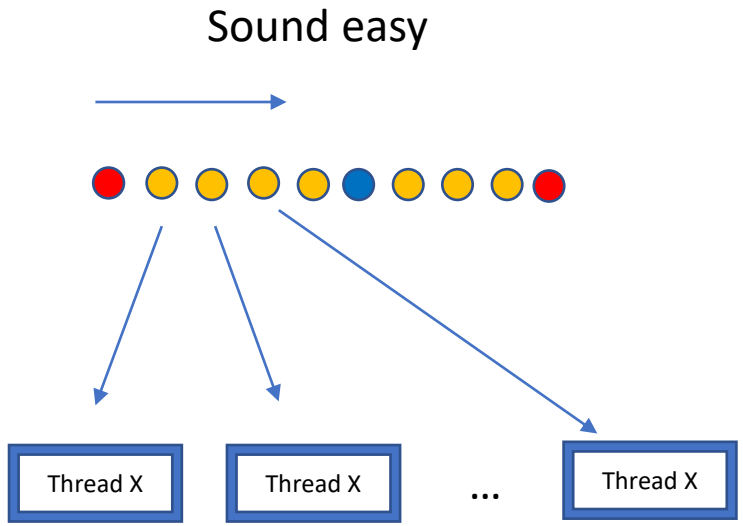


Figure 2: PRM overview and parallel components in our algorithm

Sources:
g-Planner: Real-time Motion Planning and Global Navigation using GPUs – Jia Pan and Christian Lauterbach and Dinesh Manocha – http://gamma.cs.unc.edu/gplanner/AAAI.pdf

- As with most algorithms the local planning is the most expensive task.
- Each edge has to be verified.
- For simple (in regards to geometry of the robot) workspaces continuous collision detection is used.
- For complex workspaces (practice) the discrete collision detection is used.
- For $m * k/2$ edges you have to compute resolution r collision detection calls. (at least $m * k/2$ threads)
- Parallelizing the edge computation can be done. How?

Input: Parallelizing the edge computation.



Every thread takes one of the collision detections and as soon as one configuration is invalid it calls the other threads to stop

Practice:

- Stopping all threads is not immediate.
- To stop each thread is an overhead.
- Especially in crowded configurations many configurations are stuck. Meaning, that one of the first few configurations is already in collision.→ Many threads have already started CD for configurations that are not needed. → overhead
- Often the overhead consumes the benefit of parallelizing.
- Still paralleling is beneficial, especially for high resolutions which is often needed in practice.

The Approach – Local planning

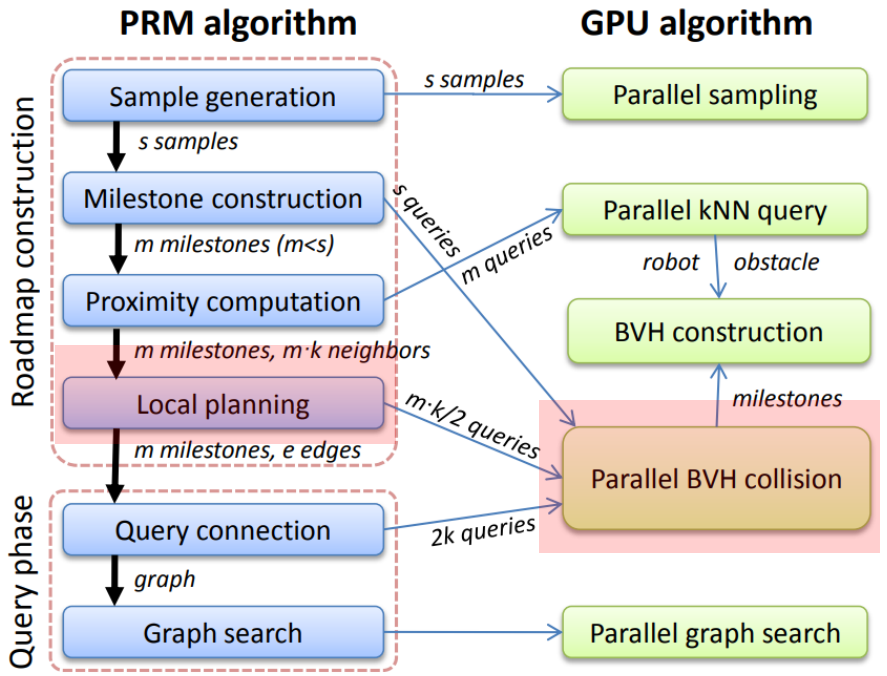
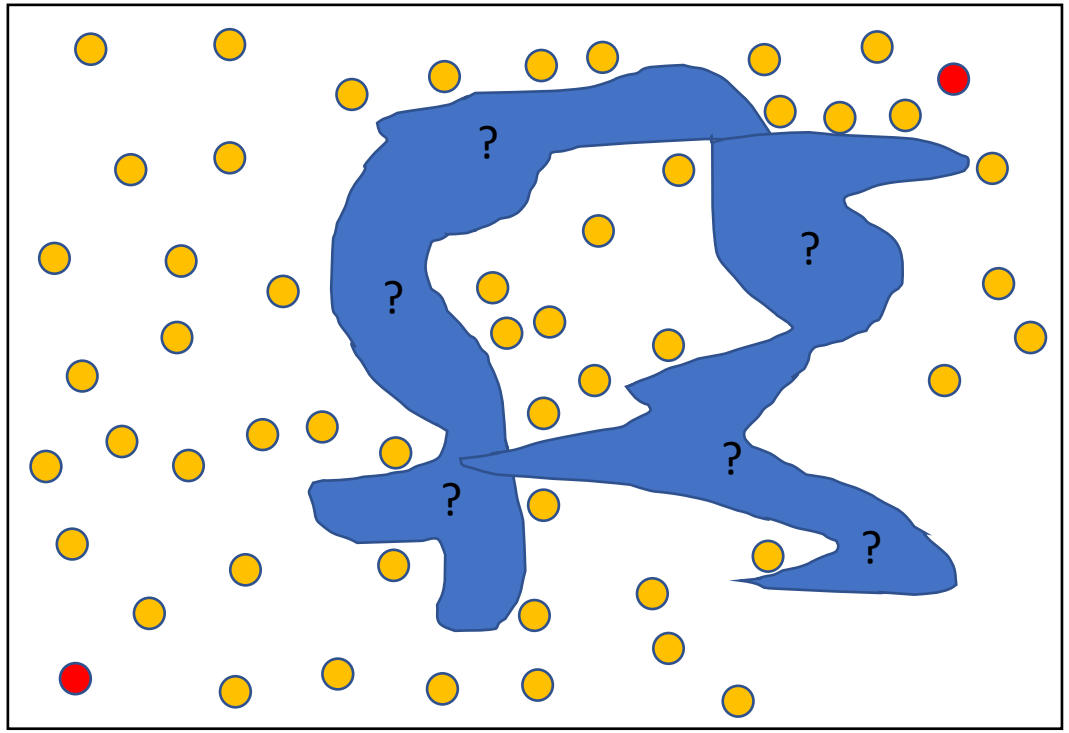


Figure 2: PRM overview and parallel components in our algorithm

- So we now know that the local planning is the most expensive task.
- Is there a method to improve performance for this?
- Yes there is and this is called lazy planning.

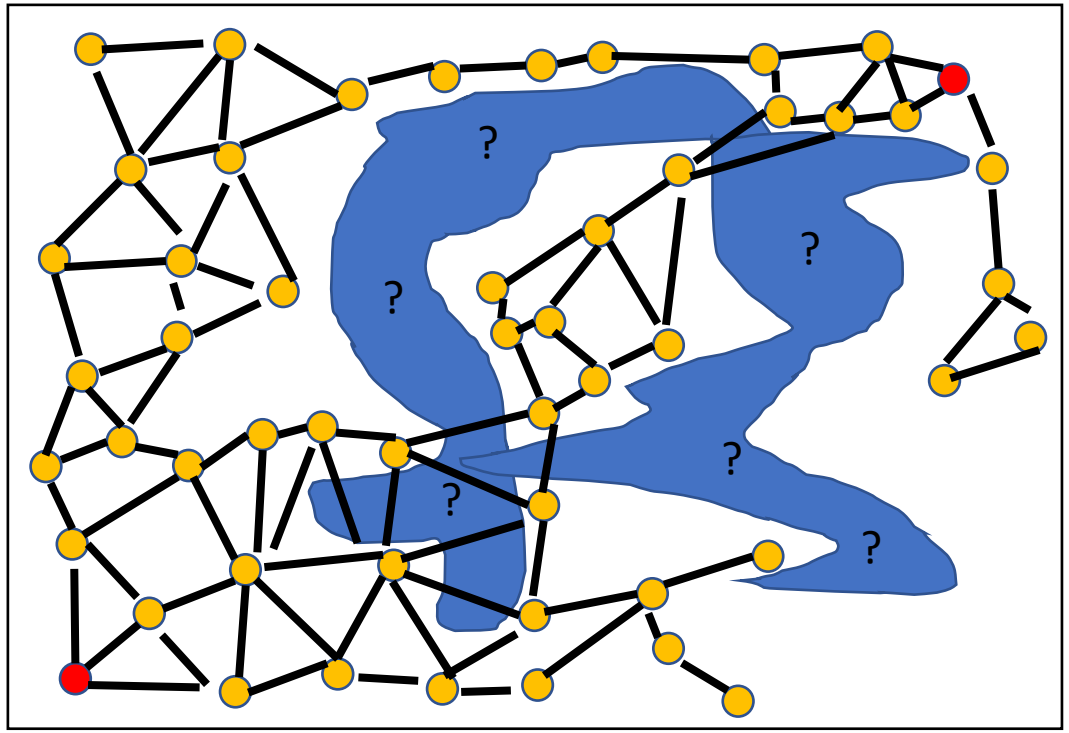
Sources:
g-Planner: Real-time Motion Planning and Global Navigation using GPUs – Jia Pan and Christian Lauterbach and Dinesh Manocha – http://gamma.cs.unc.edu/gplanner/AAAI.pdf

What is lazy planning?



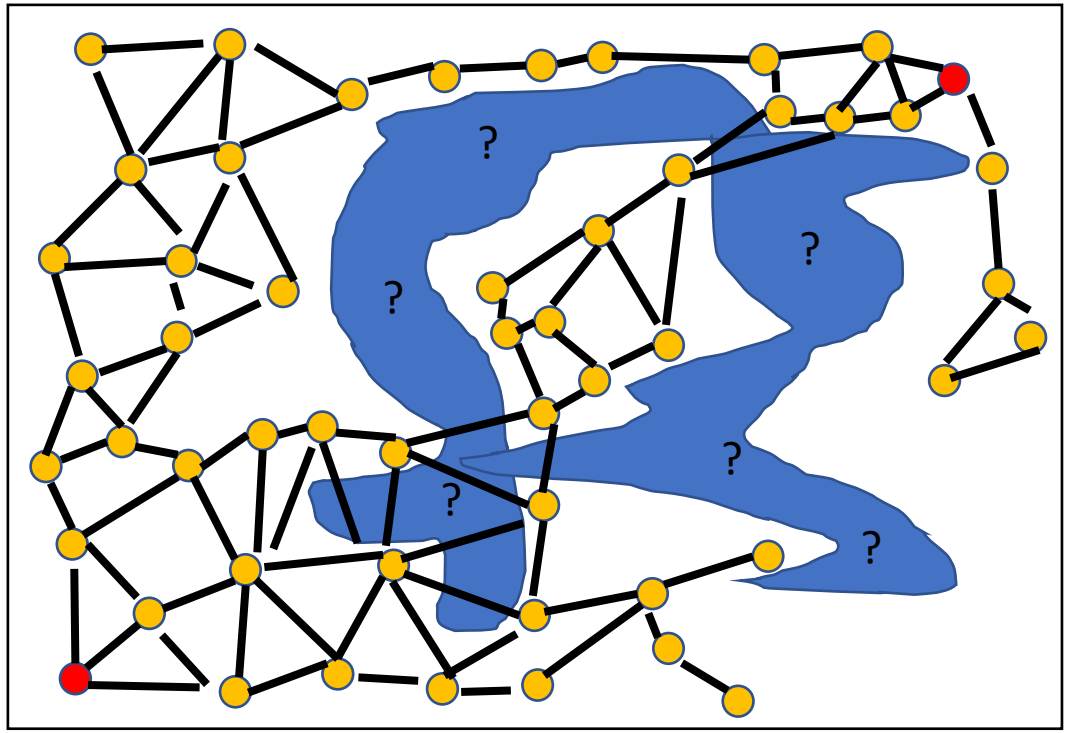
- You sample all the configurations like you do in the “normal” sPRM.

What is lazy planning?



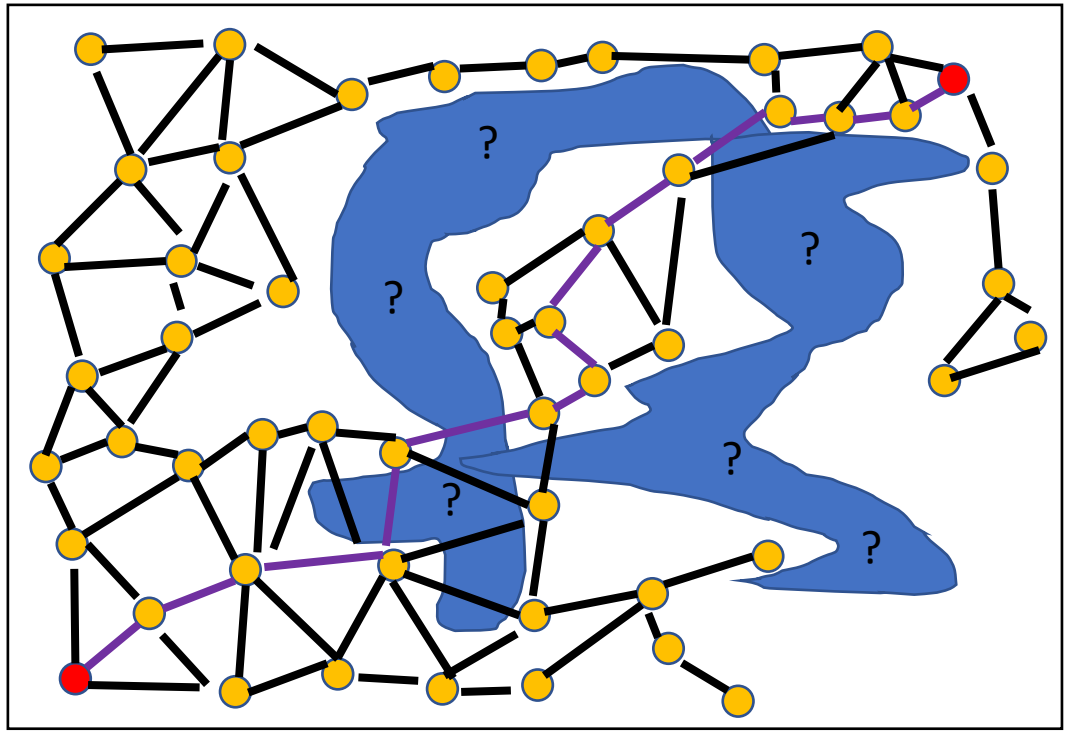
- You sample all the configurations like you do in the “normal” sPRM.
- The you compute the neighbours and draw edges between all the neighbours.

What is lazy planning?



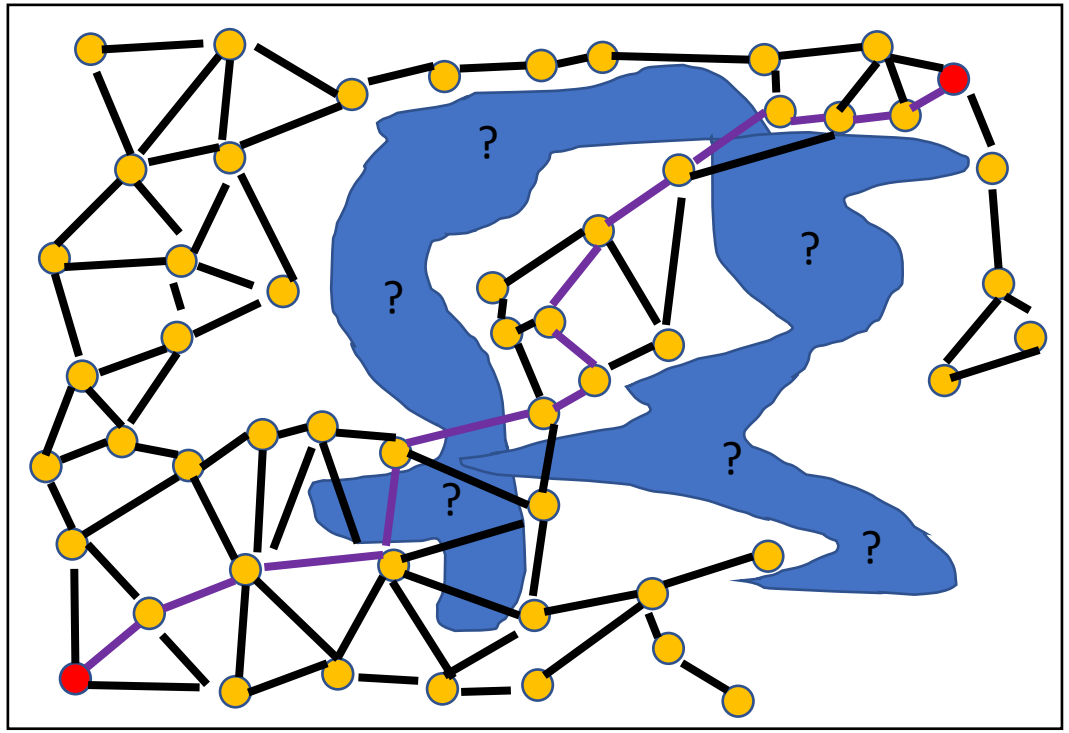
- You sample all the configurations like you do in the “normal” sPRM.
- The you compute the neighbours and draw edges between all the neighbours.
- You do NOT check if the edges are free of collision. This is what makes this algorithm lazy.

What is lazy planning?



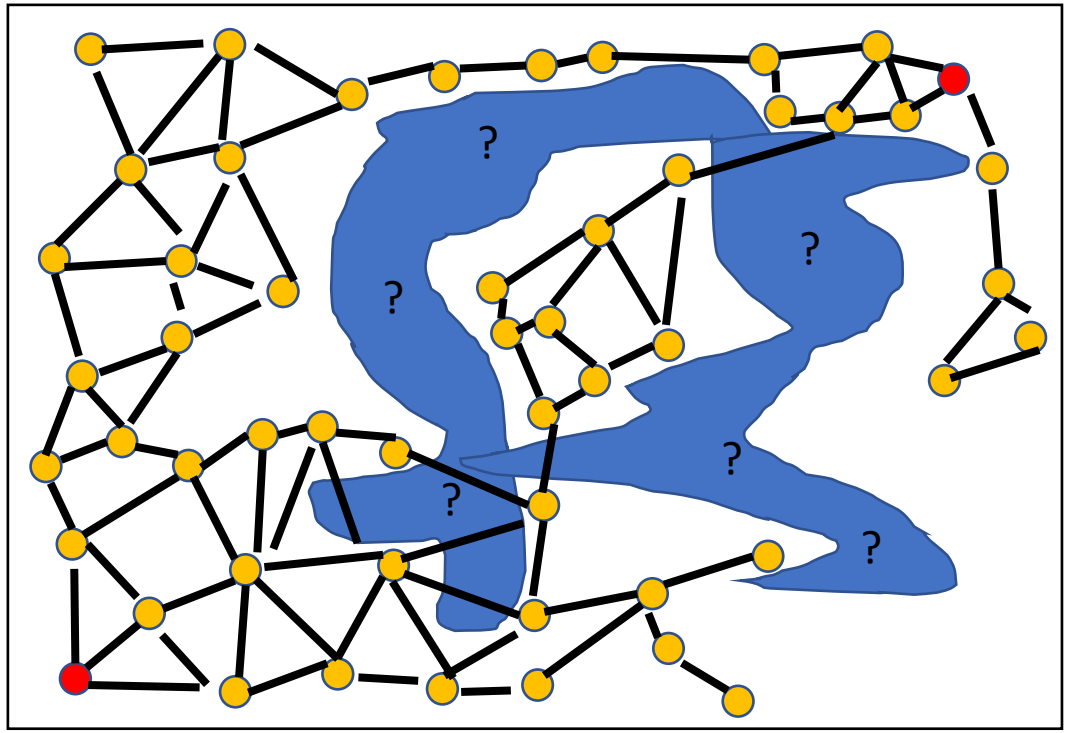
- You sample all the configurations like you do in the “normal” sPRM.
- The you compute the neighbours and draw edges between all the neighbours.
- You do NOT check if the edges are free of collision. This is what makes this algorithm lazy.
- You then compute the shortest path.

What is lazy planning?



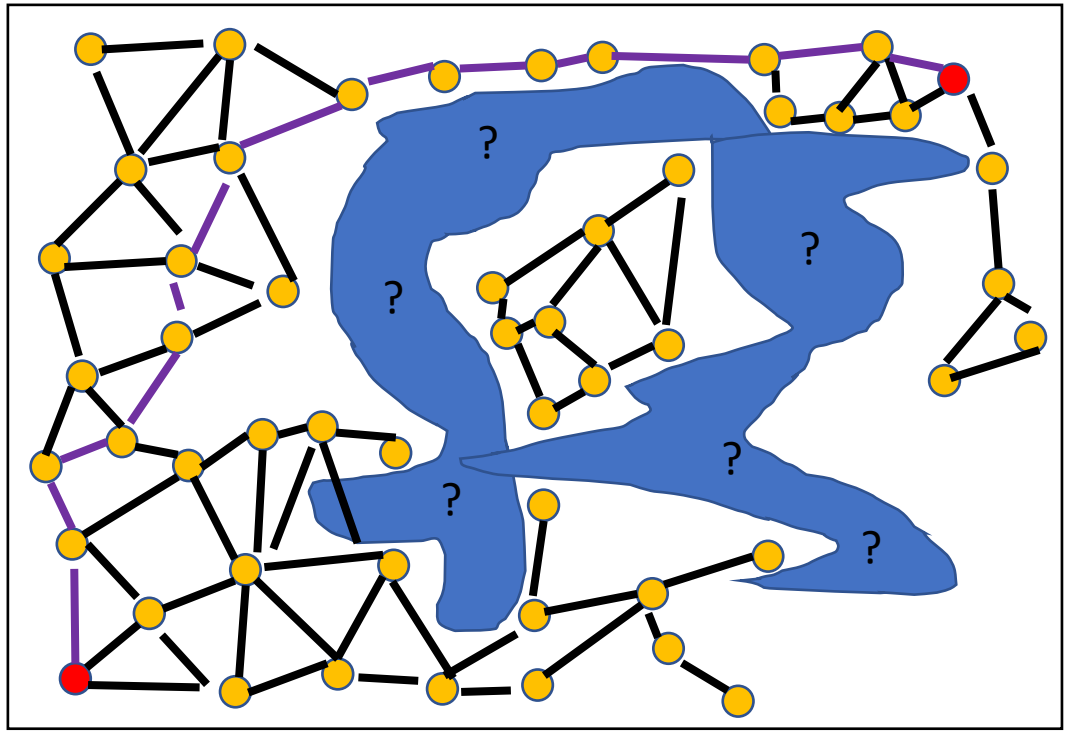
- You sample all the configurations like you do in the “normal” sPRM.
- The you compute the neighbours and draw edges between all the neighbours.
- You do NOT check if the edges are free of collision. This is what makes this algorithm lazy.
- You then compute the shortest path.
- Just now you start to check the edges for collision.

What is lazy planning?



- You sample all the configurations like you do in the “normal” sPRM.
- The you compute the neighbours and draw edges between all the neighbours.
- You do NOT check if the edges are free of collision. This is what makes this algorithm lazy.
- You then compute the shortest path.
- Just now you start to check the edges for collision.
- If there is a edge that is in collision you remove that edge (better: insert a configurations at the border)

What is lazy planning?



- You sample all the configurations like you do in the “normal” sPRM.
- The you compute the neighbours and draw edges between all the neighbours.
- You do NOT check if the edges are free of collision. This is what makes this algorithm lazy.
- You then compute the shortest path.
- Just now you start to check the edges for collision.
- If there is a edge that is in collision you remove that edge (better: insert a configurations at the border)
- Repeat this until you have found the solution path

The Approach – Local planning

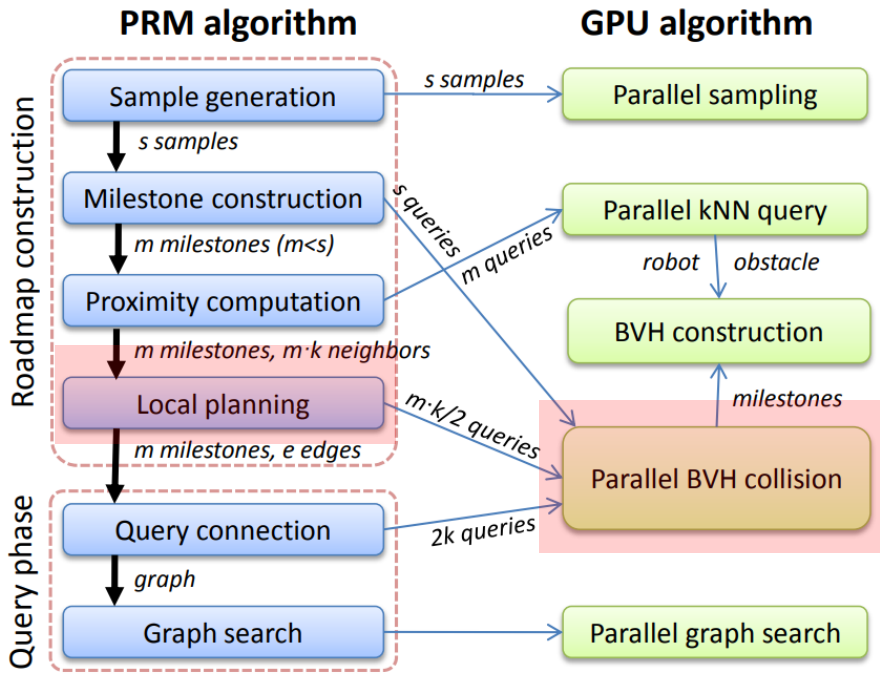


Figure 2: PRM overview and parallel components in our algorithm

- So we now know that the local planning is the most expensive task.
- Is there a method to improve performance for this?
- Yes there is and this is called lazy planning.
- This lazy planning can substantially improve performance.

Sources:
g-Planner: Real-time Motion Planning and Global Navigation using GPUs – Jia Pan and Christian Lauterbach and Dinesh Manocha – http://gamma.cs.unc.edu/gplanner/AAAI.pdf

The Approach – Query

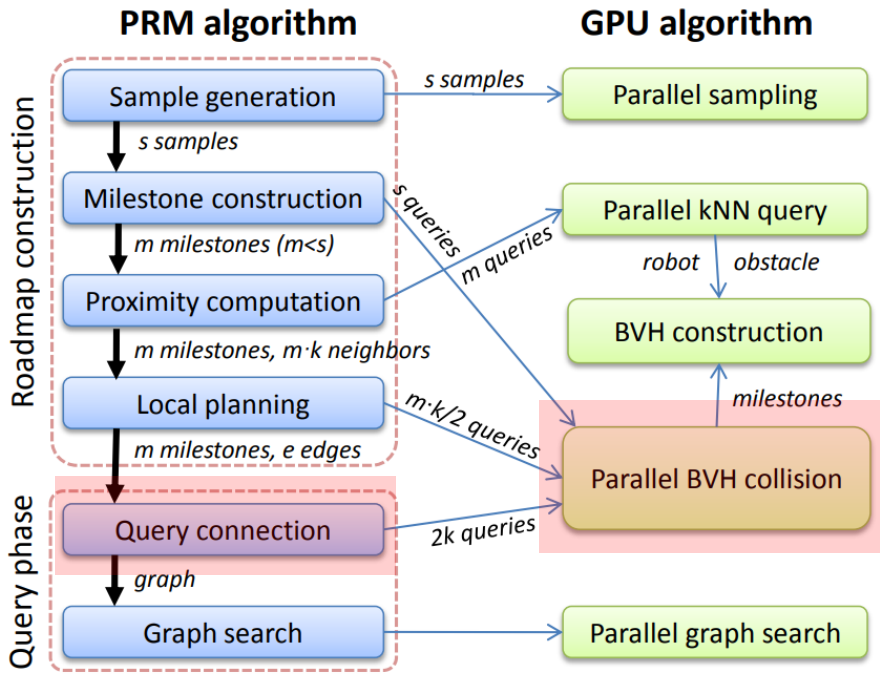


Figure 2: PRM overview and parallel components in our algorithm

- When connection the query to the roadmap the BVH is used.
- This is drawing edges to the k neighbours.
- Parallelizing the query is little beneficial, as the amount of neighbours is in most cases very low.

Sources:
g-Planner: Real-time Motion Planning and Global Navigation using GPUs – Jia Pan and Christian Lauterbach and Dinesh Manocha – http://gamma.cs.unc.edu/gplanner/AAAI.pdf

The Approach – Query

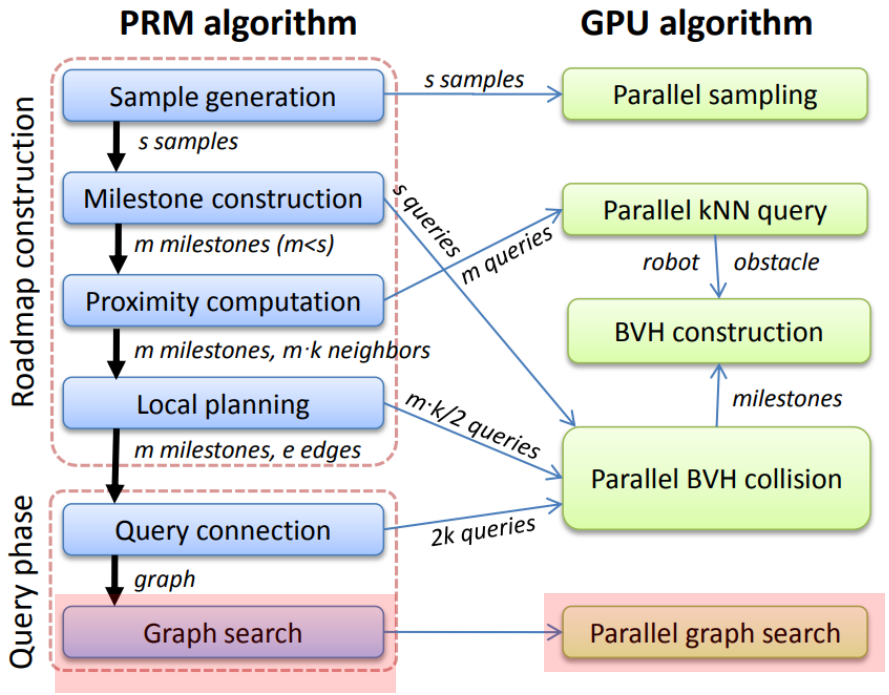


Figure 2: PRM overview and parallel components in our algorithm

- Here a parallel graph search algorithms are used.
- One approach is to go for is a parallel randomized A* algorithm.

Sources:
g-Planner: Real-time Motion Planning and Global Navigation using GPUs – Jia Pan and Christian Lauterbach and Dinesh Manocha – http://gamma.cs.unc.edu/gplanner/AAAI.pdf

The Approach – Does it work?

	C-PRM	C-RRT	G-PRM	GL-PRM
piano	6.53s	19.44s	1.71s	111.23ms
helicopter	8.20s	20.94s	2.22s	129.33ms
maze3d1	138s	21.18s	14.78s	71.24ms
maze3d2	69.76s	17.4s	14.47s	408.6ms
maze3d3	8.45s	4.3s	1.40s	96.37ms
alpha1.5	65.73s	2.8s	12.86s	1.446s

Table 1: The left two columns highlight the implementations of PRM and RRT algorithm in the OOPSMP. The right two columns highlight the performance of our GPU-based algorithms.

- The improvement are substantial.
- Especially lazy planning on a GPU proves to work wonders.
- Some note: You see on the left two columns that judging on PRM vs RTT is not possible for each example.

Sources:
g-Planner: Real-time Motion Planning and Global Navigation using GPUs
– Jia Pan and Christian Lauterbach and Dinesh Manocha –
http://gamma.cs.unc.edu/gplanner/AAAI.pdf

The A* algorithm

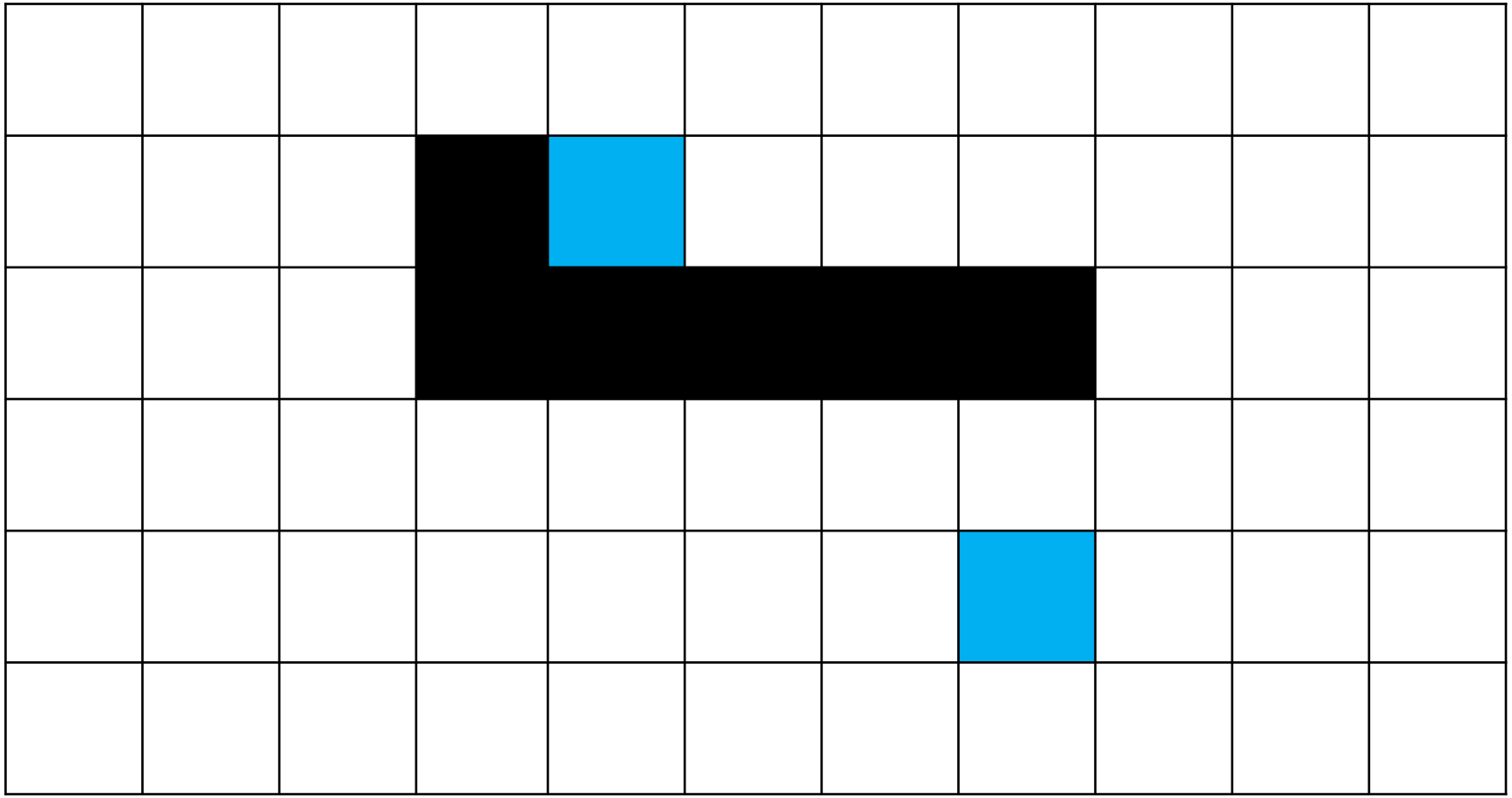
- This algorithm can be used to find optimal paths in graphs with positive weights.
- It is an extension of the Dijkstra Algorithm that was covered in AlgoDat1
- This Algorithm will find an optimal path, if there is one.

G - distance from starting node
H = distance to end node
F = G + H

G H
F

14 28
42

The A* algorithm



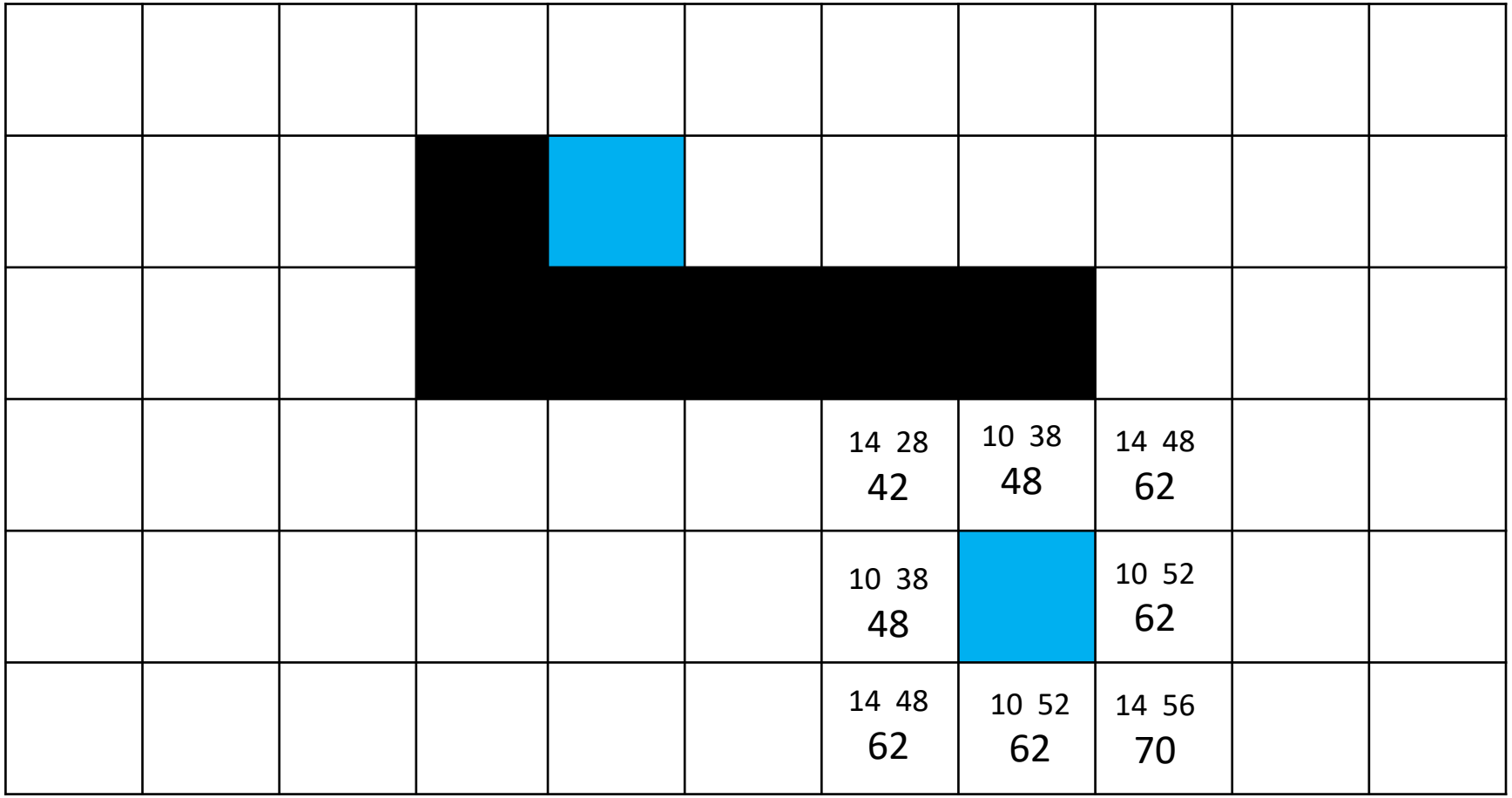
[Example taken from Sebastian Lague]

G - distance from starting node
H = distance to end node
F = G + H

G H
F

14 28
42

The A* algorithm

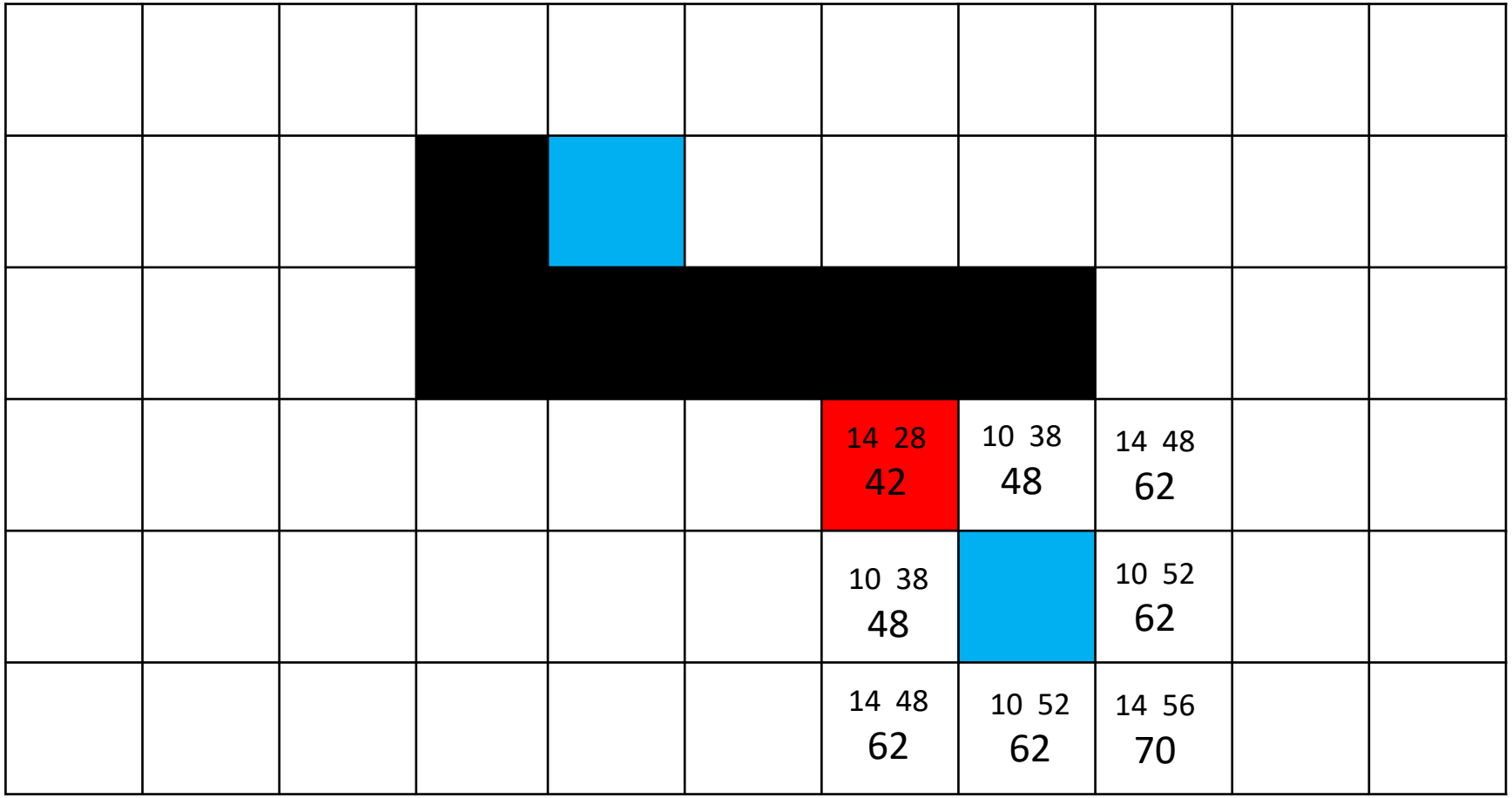


G - distance from starting node
H = distance to end node
F = G + H

G H
F

14 28
42

The A* algorithm



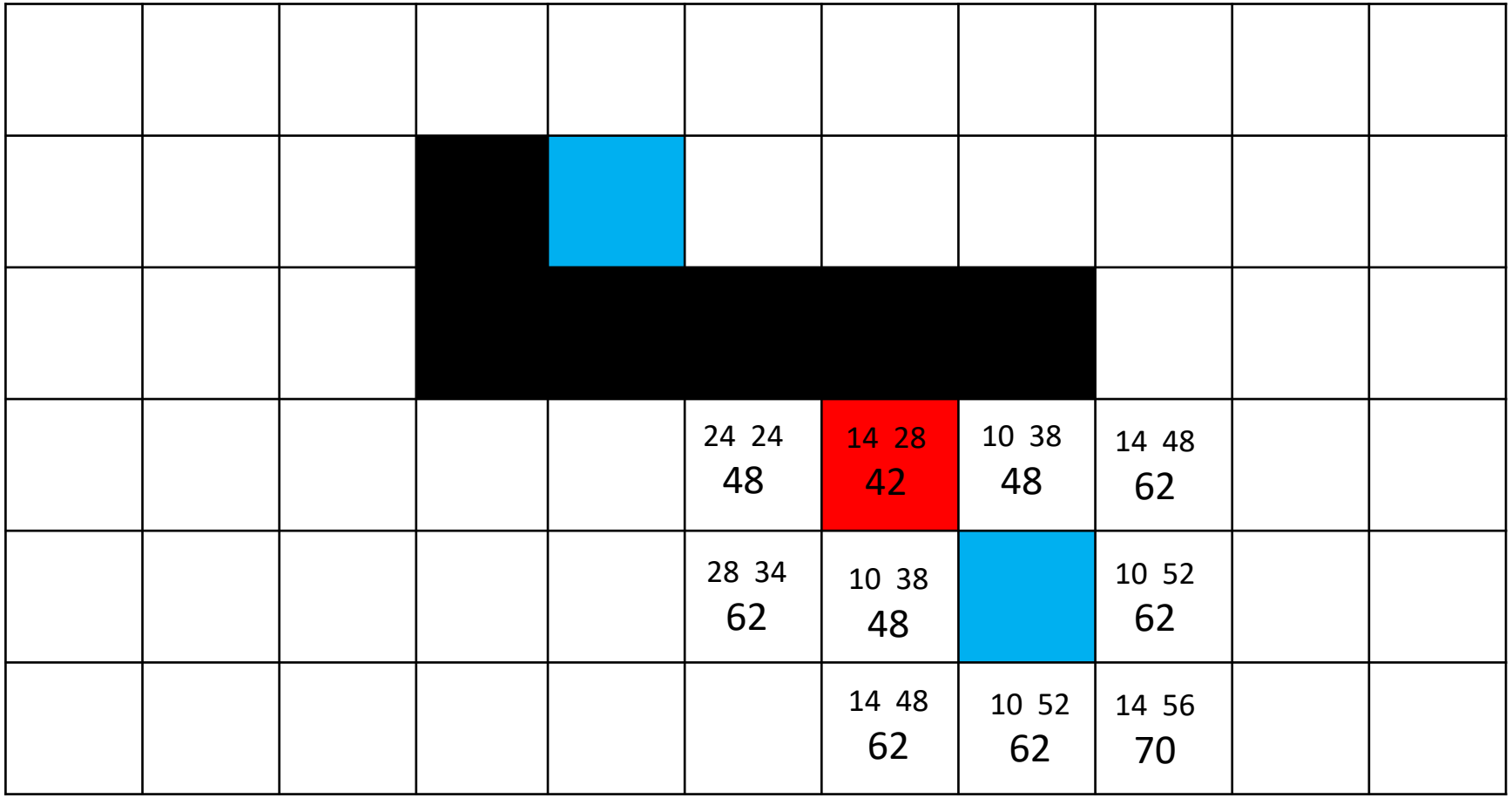
- Take the one that has the lowest F cost.

G - distance from starting node
H = distance to end node
F = G + H

G H
F

14 28
42

The A* algorithm



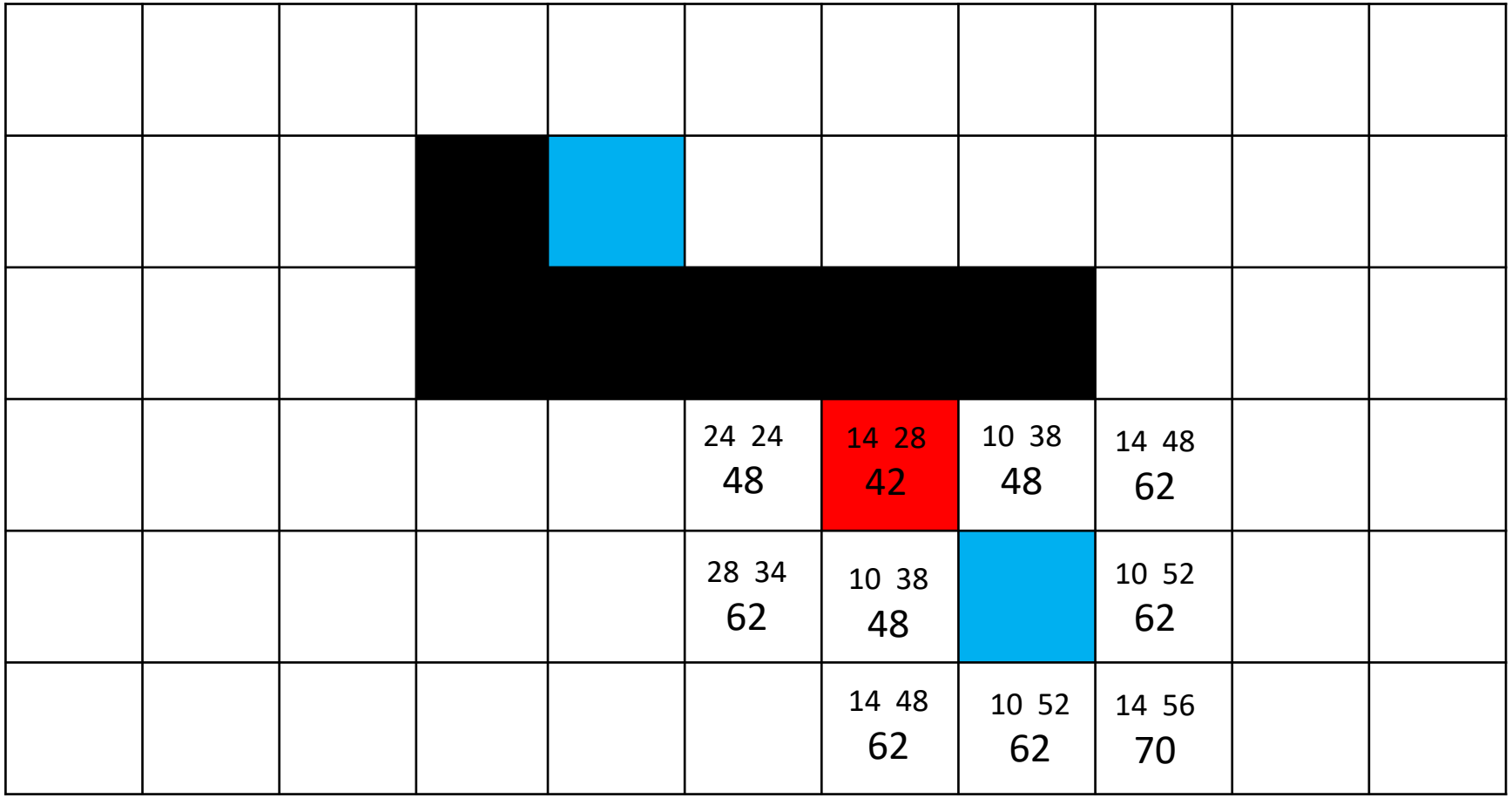
- Expand

G - distance from starting node
H = distance to end node
F = G + H

G H
F

14 28
42

The A* algorithm



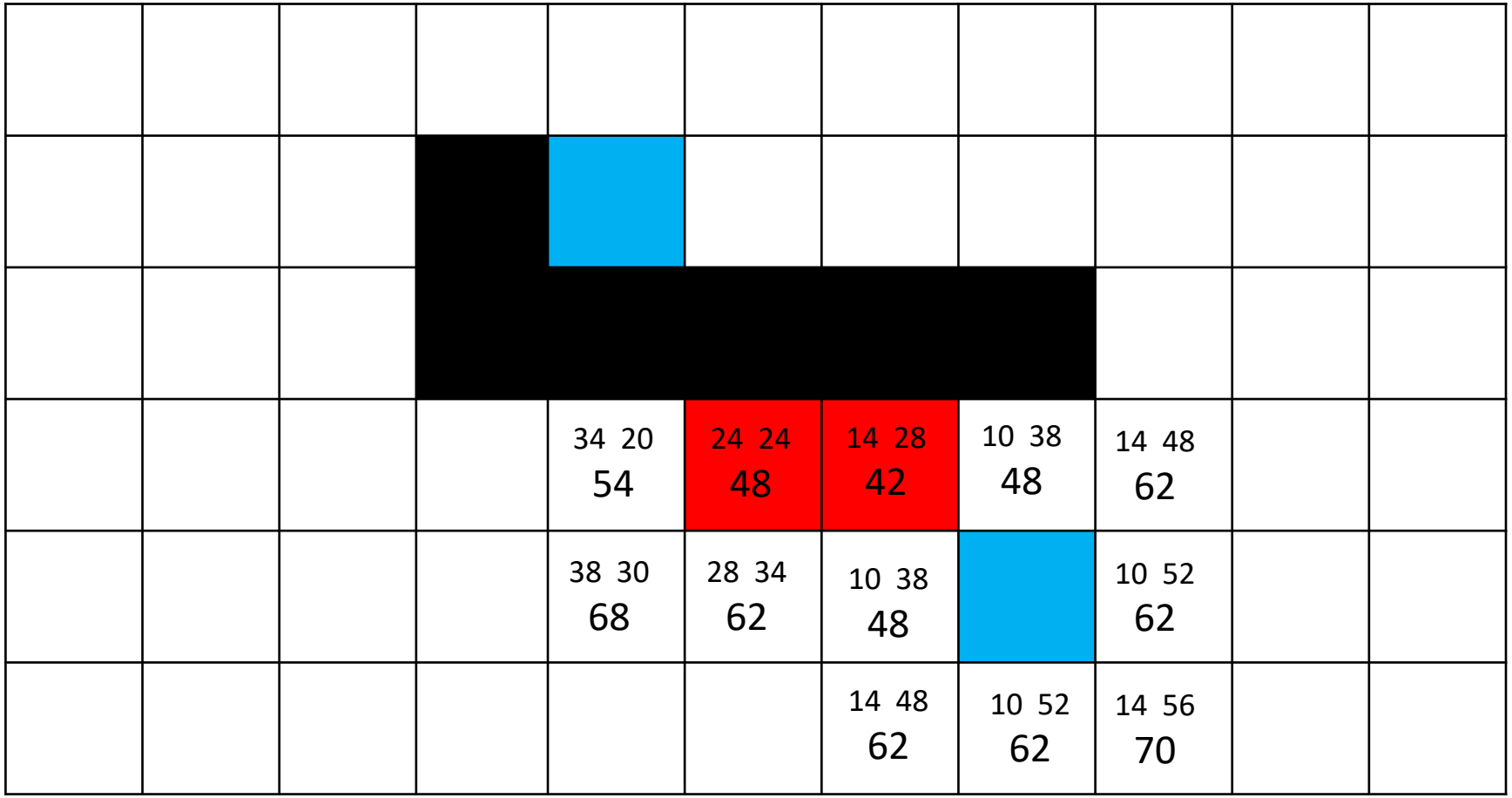
- Take again the one with the lowest.
- If two are equal, take the one with the lower H cost.

G - distance from starting node
H = distance to end node
F = G + H

G H
F

14 28
42

The A* algorithm



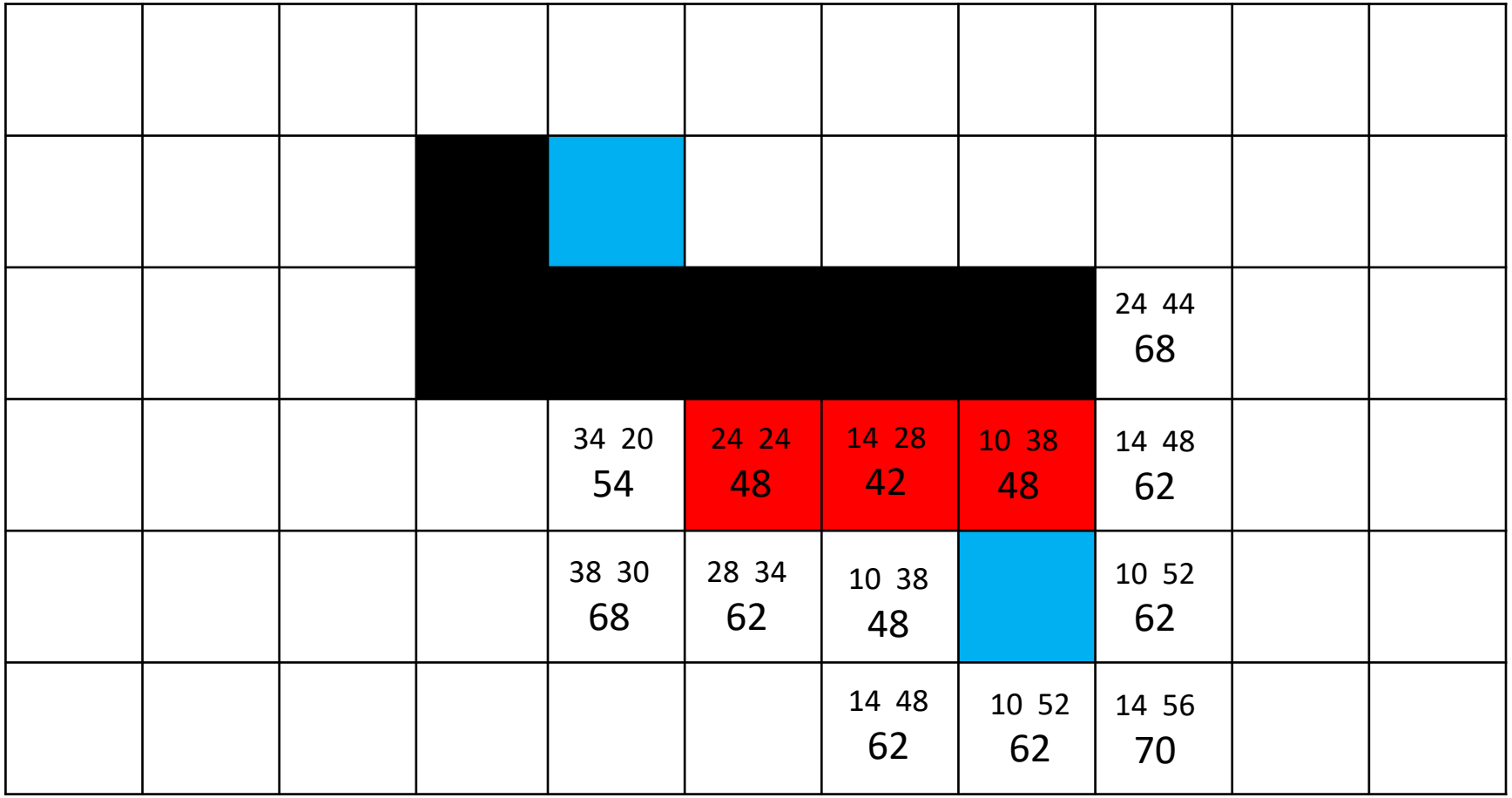
- Take again the one with the lowest.
- If two are equal, take the one with the lower H cost.
- Expand

G - distance from starting node
H = distance to end node
F = G + H

G H
F

14 28
42

The A* algorithm



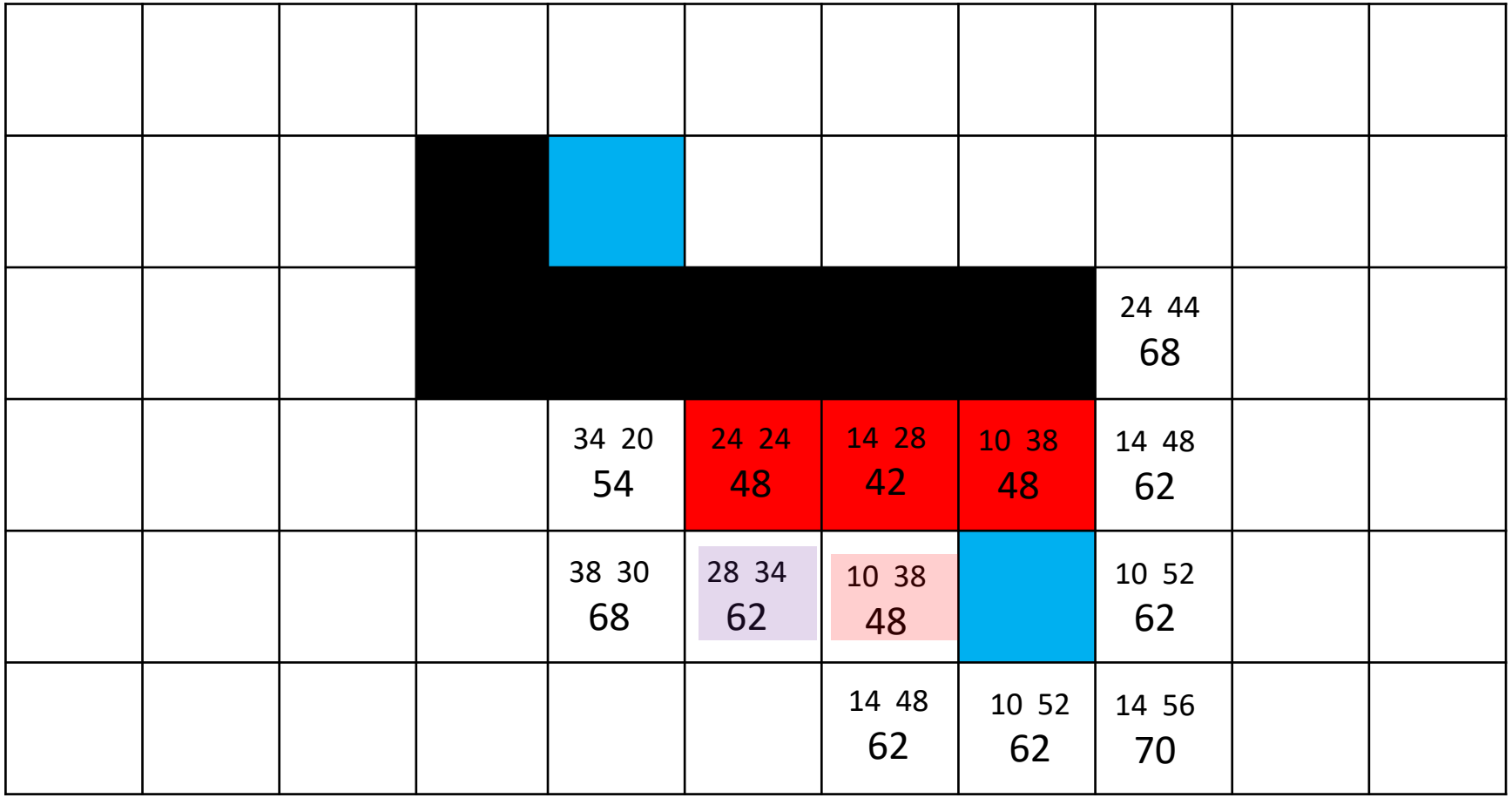
- Repeat

G - distance from starting node
H = distance to end node
F = G + H

G H
F

14 28
42

The A* algorithm

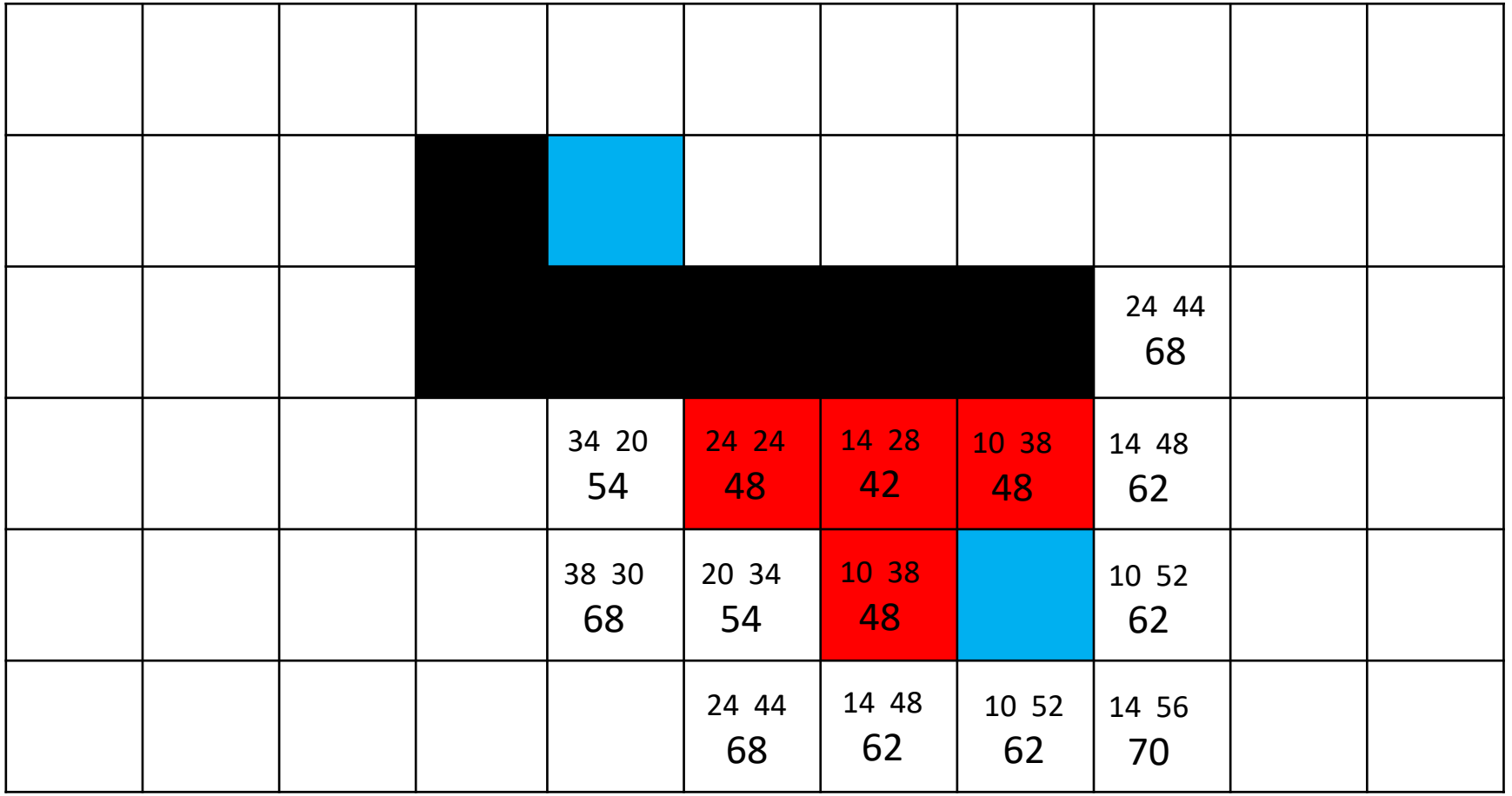


- Here we take this one.
- Note that this one here Has to be updated, as there is now a better way to get to this node.

G - distance from starting node
H = distance to end node
F = G + H

G H 14 28
F 42

The A* algorithm



- Here we take this one.
- Note that this one here Has to be updated, as there is now a better way to get to this node.

G - distance from starting node
H = distance to end node
F = G + H

G	H
F	

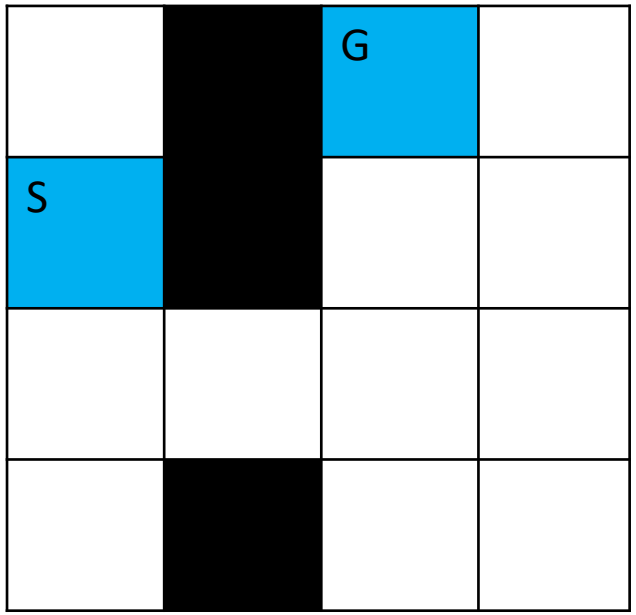
14	28
42	

The A* algorithm

				72 10 82	62 14 76	52 24 76	48 34 82	52 44 96		
				68 0 68	58 10 68	48 20 68	38 30 68	34 40 74	38 50 88	
		58 24 82						24 44 68	28 54 82	
		54 28 82	44 24 68	34 20 54	24 24 48	14 28 42	10 38 48	14 48 62	24 58 82	
		58 38 96	40 34 74	30 30 60	20 34 54	10 38 48		10 52 62	20 62 82	
			44 44 88	34 40 74	24 44 68	14 48 62	10 52 62	14 56 70	24 66 90	

- Repeat more!!

The A* algorithm – Exercise



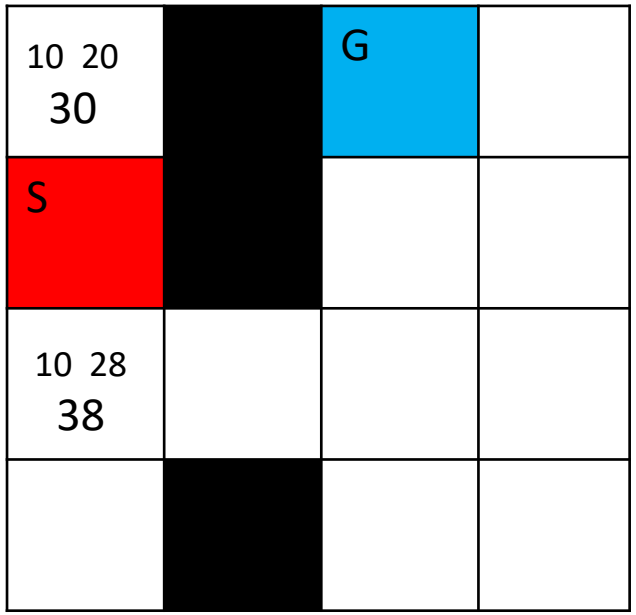
- Find the shortest path using the A* algorithm
- Note down the costs of each cell in each iteration
- Mark the cell that you have expanded.
- Stop when the shortest path has been found.

G - distance from starting node
H = distance to end node
F = G + F

G H
F

14 28
42

The A* algorithm - Exercise

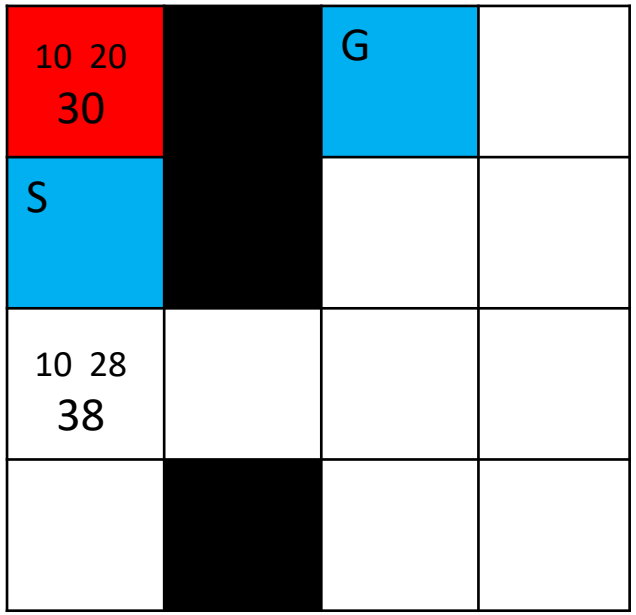


G - distance from starting node
H = distance to end node
F = G + F

G H
F

14 28
42

The A* algorithm - Exercise



G - distance from starting node
H = distance to end node
F = G + F

G H
F

14 28
42

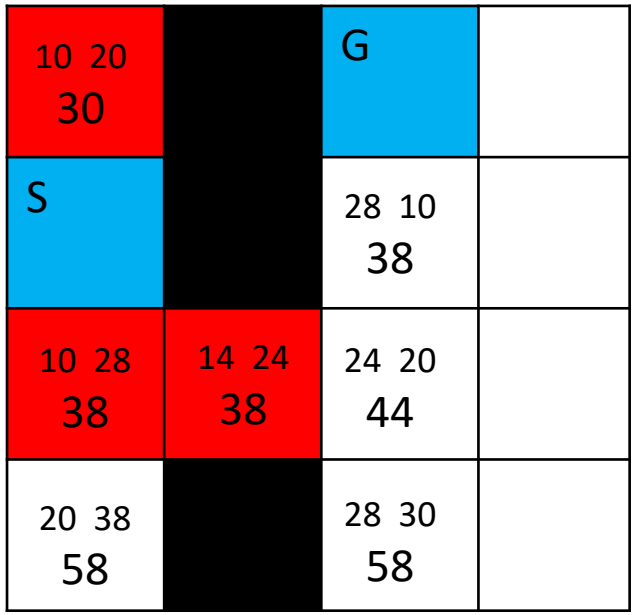
The A* algorithm - Exercise

G - distance from starting node
H = distance to end node
F = G + F

G H
F

14 28
42

The A* algorithm - Exercise

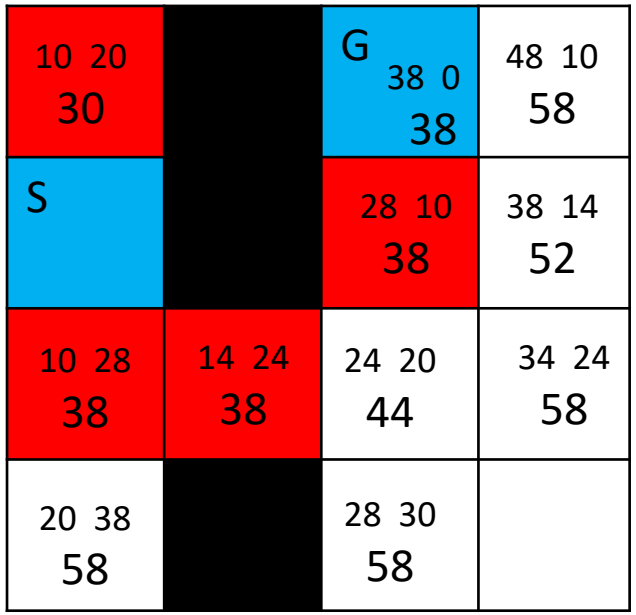


G - distance from starting node
H = distance to end node
F = G + F

G H
F

14 28
42

The A* algorithm - Exercise



For RRT

- The RRT algorithm is not easy to parallelize as the tree is a sequential data structure.
- One node depends on the remaining node.

Parallel Motion Planning using Poisson-Disk Sampling

Chonhyon Park, *Student Member, IEEE,* Jia Pan, *Member, IEEE,* and Dinesh Manocha, *Fellow, IEEE*

***Abstract*—We present a RRT-based parallel motion planning algorithm that uses the maximal Poisson-disk sampling scheme. Our approach exploits the free-disk property of the maximal Poisson-disk samples to generate nodes and perform tree expansion. Furthermore, we use an adaptive scheme to generate more samples in challenging regions of the configuration space. The Poisson-disk sampling results in improved parallel performance and we highlight the performance benefits on multi-core CPUs as well as many-core GPUs on different benchmarks.**

***Index Terms*—motion planning, Poisson-disk sampling, parallel algorithm**

I. Introduction

SAMPLING-BASED approaches are widely used to compute collision-free paths for motion planning. The most influential sampling-based motion planning schemes include probabilistic roadmaps (PRM) [1] and rapidly-exploring random trees (RRT) [2]. The key idea in these planners is to generate samples in the free configuration space of the robot and connect them with collision-free edges to construct a graph. PRM planners are mostly used for multiple-query planners and involve considerable preprocessing in terms of roadmap computation. On the other hand, most motion planning applications do not perform multiple queries. These situations arise when the robot does not know the entire environment a priori, or when it moves to a new environment. In such cases, incremental sampling-based algorithms, such as RRT, are widely used. The RRT algorithm has been extended in several aspects for use in systems with differential constraints, nonlinear dynamics, and hybrid systems. Moreover, it has also been integrated with physical robot platforms.

The simplest RRT algorithms are based on generating uniform random samples and connecting the nearby samples until a collision-free path from the initial configuration to the goal configuration has been computed. In this paper, we present a novel approach that uses Poisson-disk samples for RRT planners and constructs the trees using parallel algorithm.

Poisson-disk sampling is a well-known scheme that can be used in high dimensions to generate a random set of points with two properties: the points are tightly packed together, yet remain separated from each other by a specified minimum distance [3], [4], [5]. Poisson-disk distributions are known to have good blue-noise characteristics and are widely used in statistics, computer graphics, mesh algorithms, AI, image processing, and random object placement. Poisson-disk sampling is a sequential random process for selecting points in a region. The sampling process is *maximal* if no more points can be added, which implies that the entire region is completely covered by the disks of radius r centered at each sample.

In this paper, we present an extended version of our Poisson-RRT algorithm that originally introduced in a conference paper [6]. The algorithm uses precomputed maximal Poisson-disk samples to generate an RRT tree. We use an adaptive sampling scheme that increases the sampling rate in the challenging regions of the configuration space (e.g. narrow passages).

Main Results: The main contribution of the paper is that the use of one-time precomputation of the maximal Poisson-disk samples result in 1) fewer redundant nodes in the configuration space and 2) are more amenable to parallelization on commodity multi-core CPUs and many-core GPUs. We highlight the performance of our multi-threaded CPU and GPU-based implementations on well-known benchmarks and demonstrate improved parallelization as compared to prior parallel RRT algorithms.

The rest of the paper is organized as follows. In Section II, we survey prior work on RRT-based motion planning and highlight properties of Poisson-disk sampling. We give an overview of our Poisson-RRT planning algorithm in Section III. We present the details of the planning algorithm in Section IV and describe the parallel extension in Section V. Theoretical analysis of the algorithm is given in Section VI. We highlight the performance on different benchmarks of low- and high-dimensional spaces in Section VII.

II. Related Work and Background

In this section, we give a brief overview of prior work on RRT-based motion planning, parallel planning algorithms, Poisson-disk sampling, and Lattice-based Sampling.

A. RRT-based Motion Planning

There is extensive work on RRT-based motion planning due to its efficiency. The original RRT algorithm [7] grows the RRT tree based on the Voronoi property, biasing the search towards unexplored regions of free configuration space. Many variants to improve this original RRT have been proposed.

This work was supported by the Army Research Office under Contract W911NF-10-1-0506, by the National Science Foundation under Award 1000579, Award 1117127, and Award 1305286, and by Sandia Labs. A preliminary version of this paper was presented in part at the IEEE International Conference on Robotics and Automation, Hong Kong, China, May 2014.

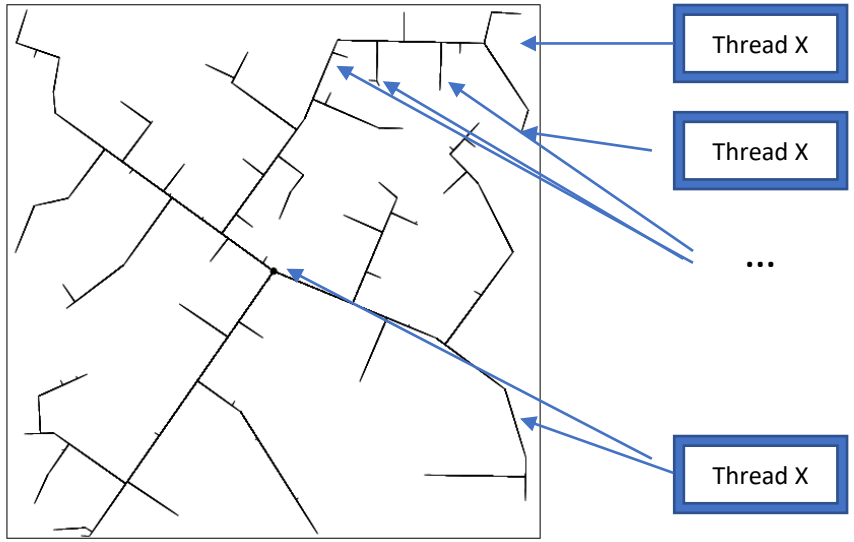
Chonhyon Park and Dinesh Manocha are with the Department of Computer Science, University of North Carolina at Chapel Hill, Chapel Hill, NC 27599 USA (E-mail: {chpark, dm}@cs.unc.edu).

Jia Pan is with the Department of Computer Science, University of Hong Kong, Pokfulam, Hong Kong (E-mail: jpan@cs.hku.hk).

Sources:
g-Planner: Real-time Motion Planning and Global Navigation using GPUs – Chonhyon Park, Jia Pan, and Dinesh Manocha,– http://gamma.cs.unc.edu/PoissonRRT/PoissonRRT_TRO.pdf

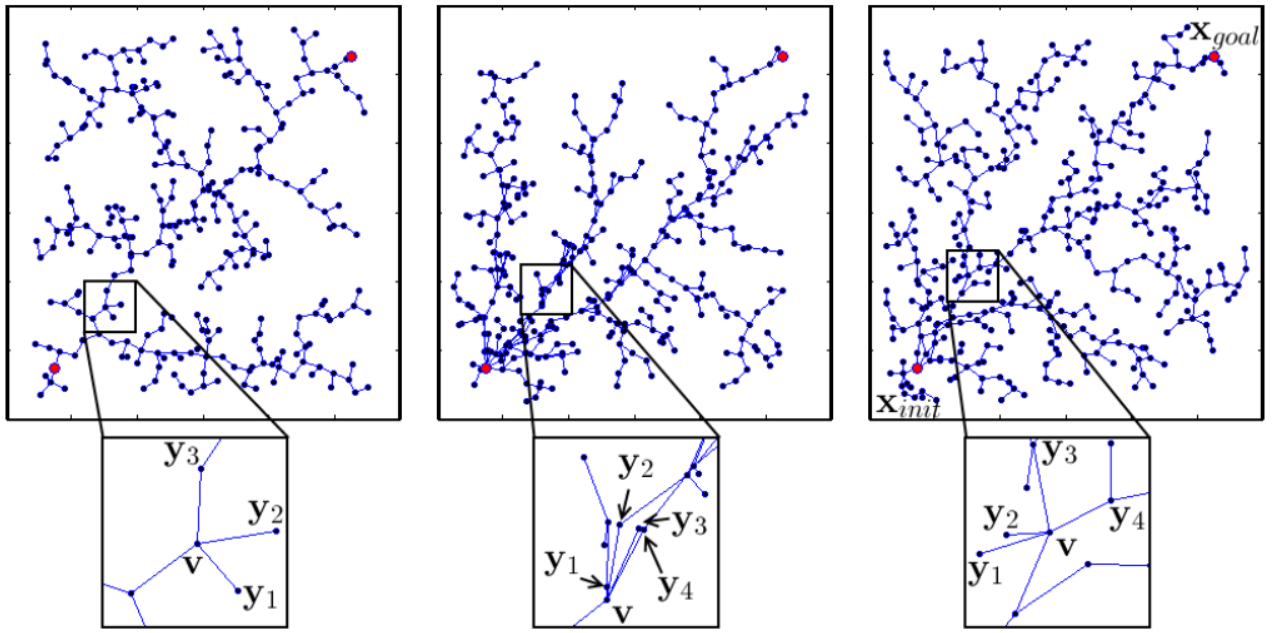
The problem

- Plain parallelizing of the edge computation leads to a clustering of tree branches. The tree is no longer expanding efficient → Improvement of multiple cores is lost.



The problem

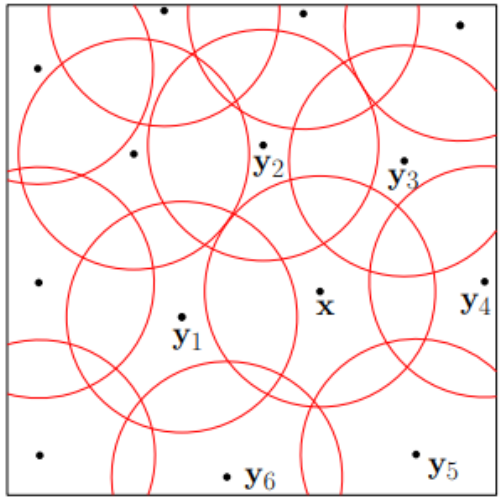
- This leads to a tree that is not clean and does not keep the Voronoi property of the sampling.



(a) RRT (b) AND Parallel RRT (c) Parallel Poisson-RRT

Fig. 3. Comparison of RRT trees generated using different planning approaches. (a) The tree corresponding to the original RRT algorithm is generated according to the Voronoi bias of the sequential algorithm. (b) The parallel RRT tree generated by AND parallelism has many redundant nodes that are close to other nodes in the tree (e.g., the new nodes $\mathbf{y}_2$, $\mathbf{y}_3$, and $\mathbf{y}_4$ are close to $\mathbf{y}$). (c) The tree generated with Poisson-disk sampling has fewer redundant nodes due to the free-disk property of samples, although it is generated using the parallel sampling.

How to solve this?

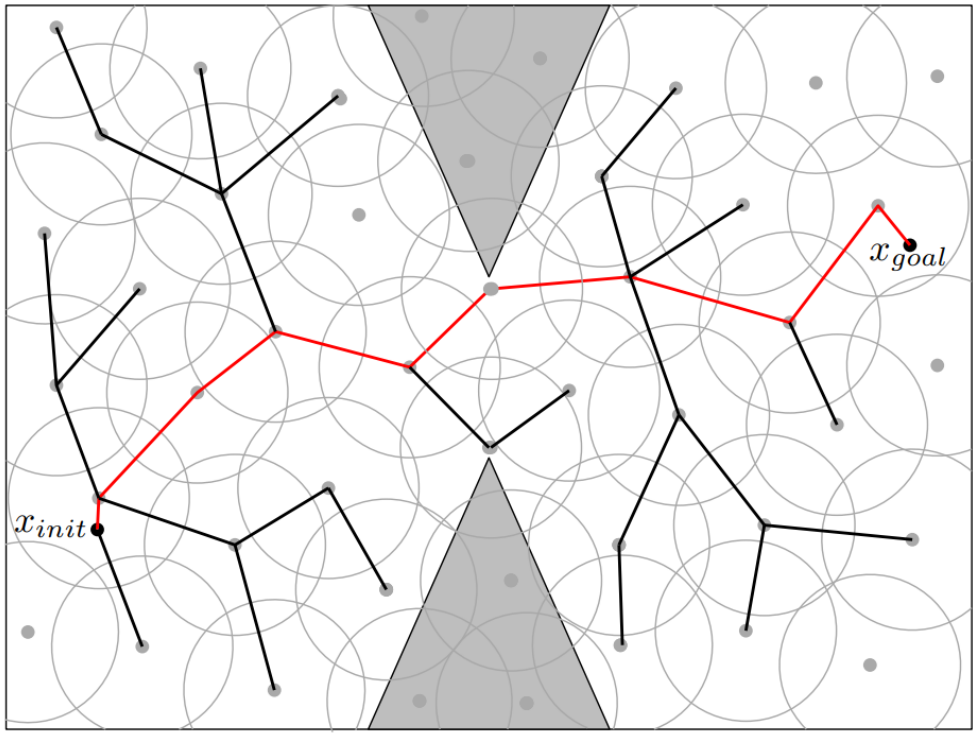


(a) Maximal Poisson-disk samples

Main Idea:

- Make sure, that each thread is not “working” in the same area/direction.
- For this you have to compute samples that have the property that the distance to its next sample is at least r.
- With this, each samples has disk associated to it. This disk (the socalled poisson disk) has at least a radius of r. Inside each disk no additional samples can be placed.
- You have to compute samples until the whole configuration space is covered.

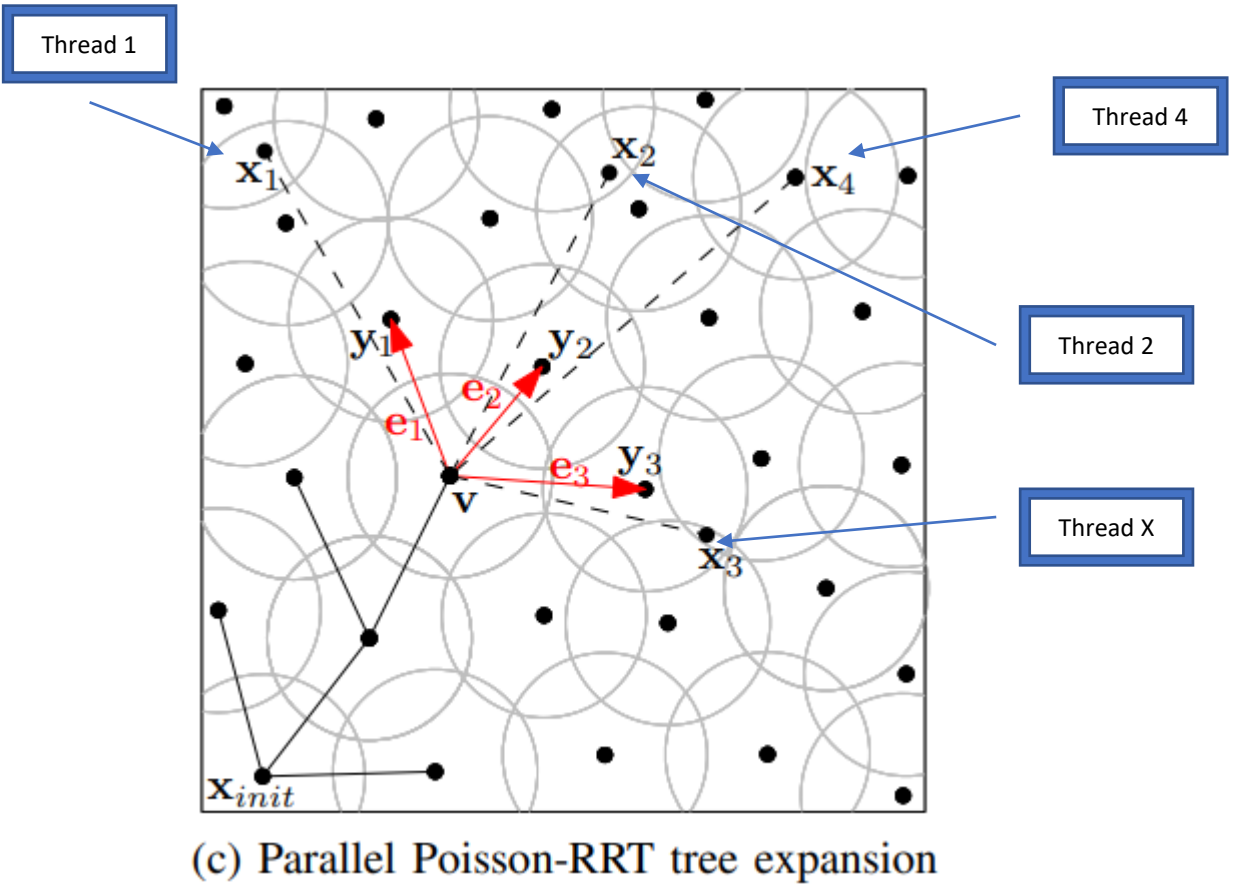
The tree..



(b) Motion planning using Poisson-disk sampling

- In the RRT the poisson disk samples are now used to build the tree.
- Each thread now takes one of the samples.
- Any other thread will not be able to compute a edge that is similar to the one computed by a other thread. Why?

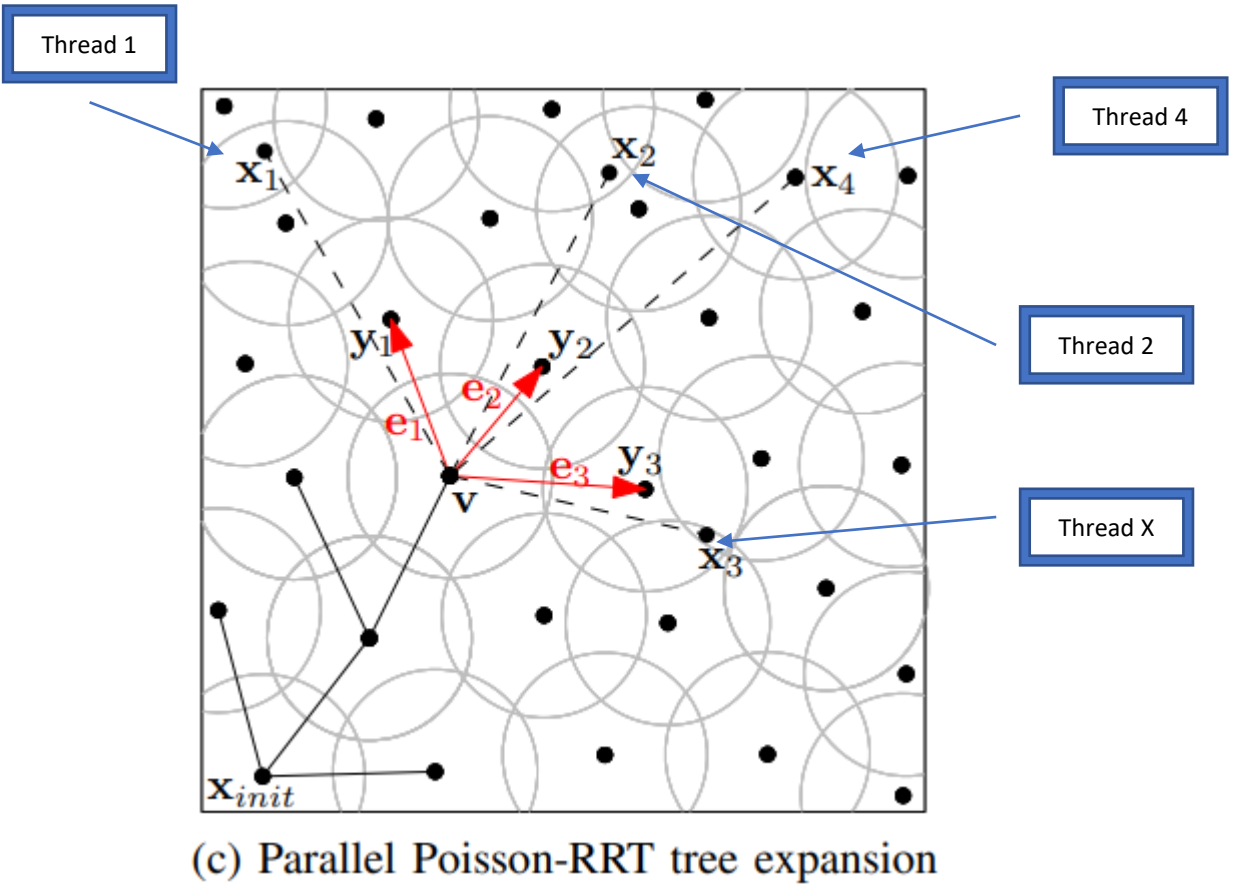
How do the threads extend the tree?



(c) Parallel Poisson-RRT tree expansion

- Thread 1 samples a configuration.

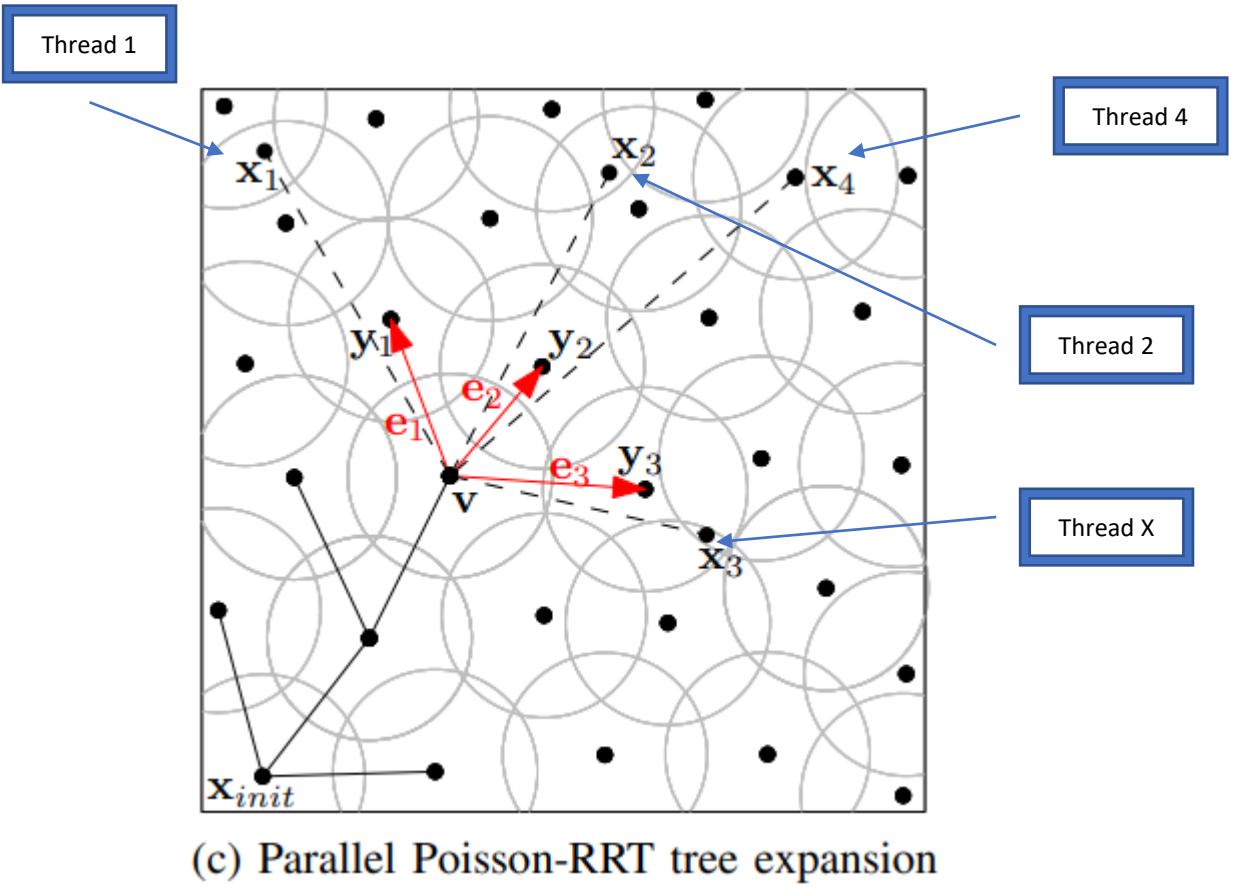
How do the threads extend the tree?



(c) Parallel Poisson-RRT tree expansion

- Thread 1 samples a configuration.
- The closest vertex to this sample is v

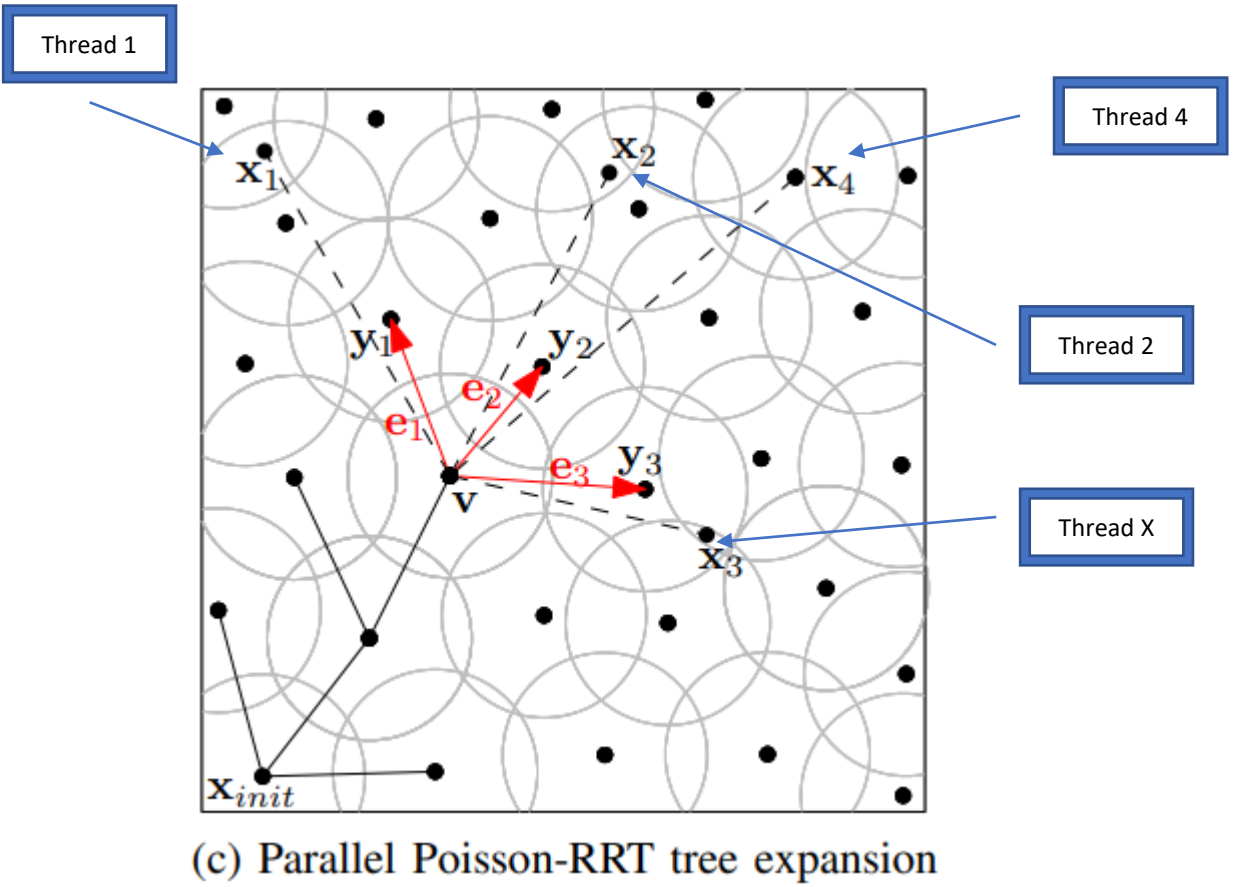
How do the threads extend the tree?



(c) Parallel Poisson-RRT tree expansion

- Thread 1 samples a configuration.
- The closest vertex to this sample is v
- Same goes for all other threads (in this example)

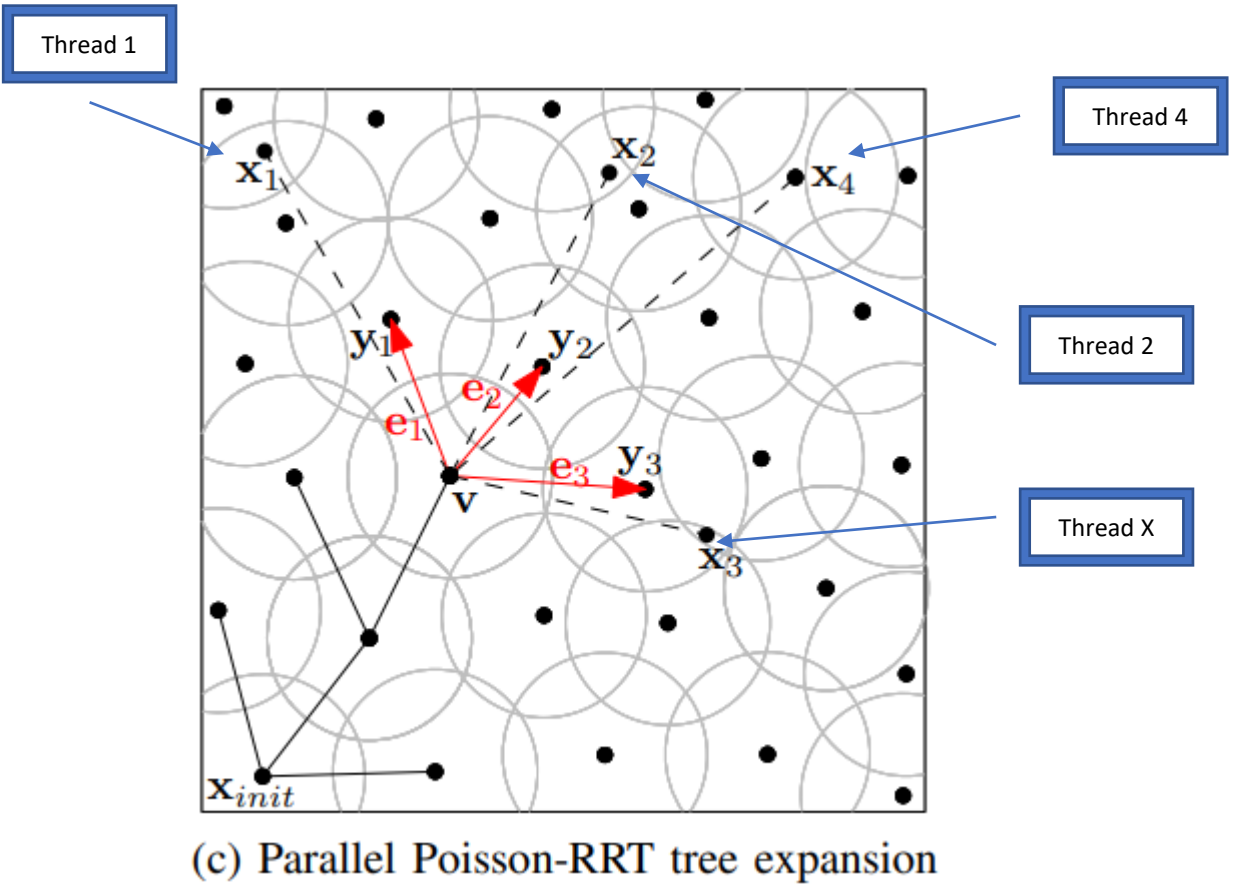
How do the threads extend the tree?



(c) Parallel Poisson-RRT tree expansion

- Thread 1 samples a configuration.
- The closest vertex to this sample is v
- Same goes for all other threads (in this example)
- Each thread checks for the closest poisson disk sample to v.

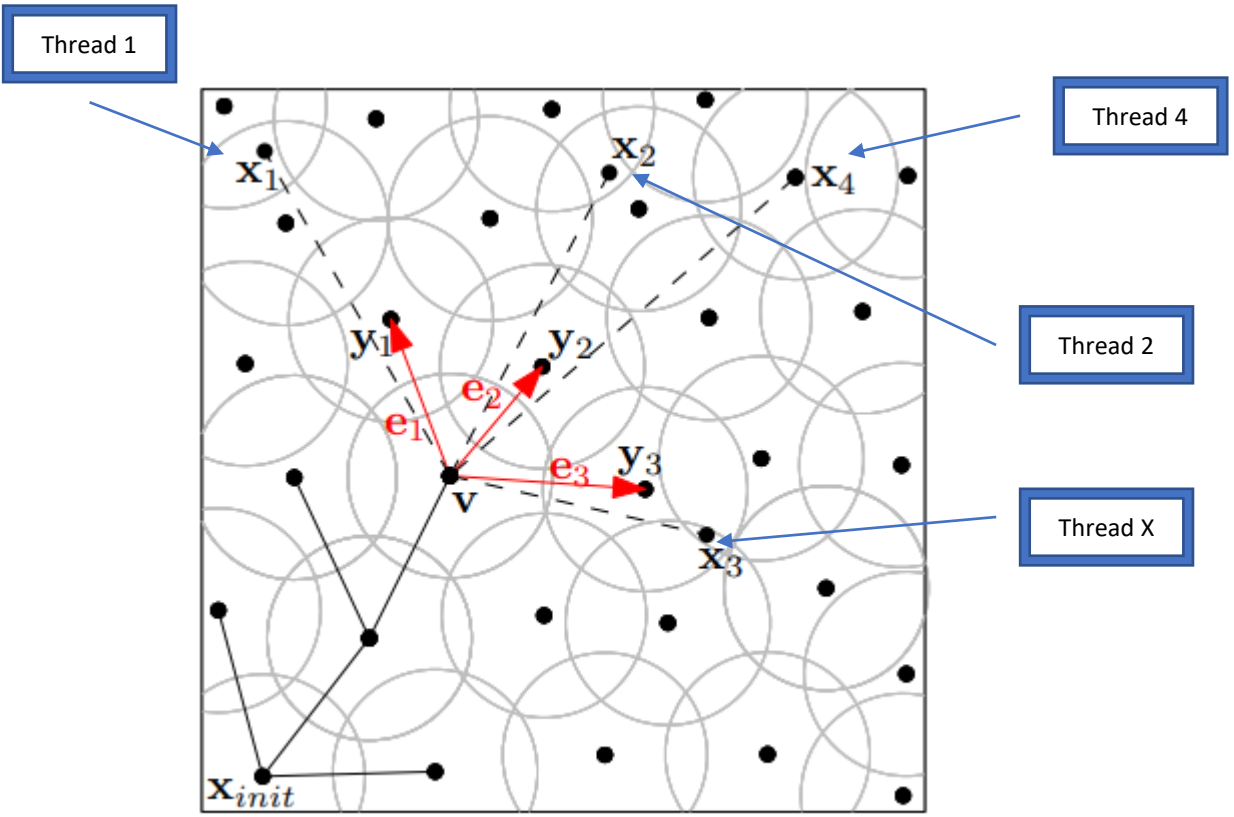
How do the threads extend the tree?



(c) Parallel Poisson-RRT tree expansion

- Thread 1 samples a configuration.
- The closest vertex to this sample is v
- Same goes for all other threads (in this example)
- Each thread checks for the closest poisson disk sample to v.
- For thread 2 and 4 this is the same. Only the one that has “finished earlier” is allowed to add the edge.

How do the threads extend the tree?



(c) Parallel Poisson-RRT tree expansion

- Thread 1 samples a configuration.
- The closest vertex to this sample is v
- Same goes for all other threads (in this example)
- Each thread checks for the closest poisson disk sample to v.
- For thread 2 and 4 this is the same. Only the one that has “finished earlier” is allowed to add the edge.
- This make sure that no similar edges (which decrease performance) are added to the tree.

One note on the samples…

- Computing the poisson disk samples is not easy. This takes time, especially for small r.
- But the disks can be computed in a preprocessing step and can be reused for other computations. (Why? The samples are independent from the workspace).
- But is this effort worth it?

The Approach – Does it work?

	CPU-based								GPU-based (32 threads)			
# of threads	single-threaded								8 threads		32 threads	
Algorithm	RRT		RRT-Connect		LazyRRT		Poisson-RRT		pRRT		pRRT	Poisson-RRT
Benchmark	Mean	Std.dev.	Mean	Std.dev.	Mean	Std.dev.	Mean	Std.dev.	Mean	Std.dev.	Mean	Std.dev.	Mean	Std.dev.
Easy	0.34	(0.33)	0.12	(0.14)	0.12	(0.09)	0.37	(0.48)	0.18	(0.15)	0.04	(0.04)	0.03	(0.03)
Cubicle	2.31	(0.84)	0.53	(0.09)	81.54	(43.07)	4.03	(1.49)	0.59	(0.31)	0.63	(0.35)	0.31	(0.36)
AlphaPuzzle	32.76	(13.54)	19.92	(14.73)	72.72	(71.74)	27.23	(27.83)	6.69	(5.28)	1.93	(1.22))	1.31	(1.28)
Apartment	232.24*	(89.42)	20.15	(20.74)	11.55	(12.18)	72.54	(62.01)	126.68	(69.94)	19.97	(7.33)	11.88	(7.95)

TABLE I

PERFORMANCE OF RRT-BASED PLANNING ALGORITHMS ON DIFFERENT BENCHMARKS. WE REPORT PLANNING TIME FOR EACH CASE. THE MEAN AND STANDARD DEVIATION ARE COMPUTED FROM 100 TRIALS ON EACH BENCHMARK. CPU-BASED PRRT UTILIZES 8 THREADS TO FULLY EXPLOIT THE 8-CORE CPU. GPU-BASED ALGORITHMS USE 32 THREADS FOR THE COMPUTATION. *RRT ALGORITHM CANNOT FIND SOLUTION IN SOME INSTANCES AND THOSE ARE TAKEN IN ACCOUNT IN COMPUTING THE AVERAGE.

- This shows that with the poisson disk samples you can improve the performance.

Summary

- There are ways to parallelize motion planning.
- Depending on the motion planning problem, the result can be impressive.
- The implementations are no longer “simple”.
- **In practice:** There is quite some knowledge needed to implement those algorithms efficiently.
- For GPU implementations you often end up in hardware dependency (e.g CUDA) if you want the optimal performance.