2007 IEEE International Conference on
Robotics and Automation
Roma, Italy, 10-14 April 2007

A Practical Pursuit-Evasion Algorithm: Detection and Tracking

Amna AlDahak and Ashraf Elnagar

Abstract— **This paper presents a practical algorithm for evader detection and tracking using one or more pursuers. The solution employs two advanced data structures. The first one is the Rapidly-Exploring Random Tree (RRT). It is constructed randomly but evenly distributed to generate a roadmap that captures the connectivity of the free space. The second data structure is the k-dimensional tree (Kd-Tree). Upon completion of the RRTs construction, their vertices are inserted in a Kd-Tree. At the tracking phase, the Kd-Tree will be queried repeatedly for retrieving the set of potential locations to be used by each pursuer in order to monitor or track an evader. Thus, the usage of the Kd-Tree will reduce the querying cost, during tracking, to a logarithmic time. A lazy collision detection strategy is used to resolve collisions with obstacles at runtime. As a result unnecessary checks are eliminated and hence improving the system performance. Simulation results show the validity of the proposed algorithm.**

I. INTRODUCTION

A crucial issue in many security, surveillance and monitoring systems is observing (tracking) the movements of targets in a bounded environment. This class of problems is known as the *pursuit-evasion* problems. Pursuit evasion is involved in many real life applications. In medical applications, one might need to move a camera around a surgery site to keep the surgeon hands under continuous observation while he moves to pick up the instruments and operates during the surgery. The camera should move it in a way to avoid any obstruction. In general the purpose of tackling this family of problems (pursuit-evasion) is to design autonomous mobile robots for applications such as surveillance, search-and-rescue, and others.

Pursuit-evasion is a motion planning problem where visibility constraints and obstacle avoidance must be taken into account simultaneously. The problem of motion planning has been recognized for many decades, but its real start was fueled by the introduction of the idea of the *configuration space* by Lozano-Perez and Wesley in 1979, [9]. The idea is to reduce the robot to a point and enlarge the obstacles in the environment accordingly. The resulted space is denoted by the C-space. The free subset of the C-space is the free space (C_{free}) and the space occupied by obstacles is the obstacle space (C_{obs}). This reduces the problem from motion planning to path planning, where the holonomic, and differential constraints are not considered. The complexity of the path planning problem increases as the dimensions of C-space grow. Most of the path planning algorithms that were proposed after the introduction of the C-space require explicit construction of C_{obs}. Some of the algorithms presented are very efficient for planning paths with very few dofs, but their performance dramatically degrades as the number of dofs increases.

The complexity of computing C_{obs} have made finding *practical* solutions for real life motion planning problems using combinatorial algorithms impossible. This has directed research efforts toward new directions and has started the ongoing evolution of the *randomized sampling-based* motion planning algorithms. The main purpose of this class of algorithms is to avoid constructing an explicit representation of the C_{obs}. Moreover, sampling-based path planning algorithms have proven to be more successful in solving path planning problems with high dofs and with holonomic and differential constraints. Sampling-based algorithms are mainly classified as *multiple-query* planners and *single-query* planners. Multiple-query planners have two main phases, the *preprocessing* phase and the *querying* phase. In the first phase the planner constructs a roadmap that will capture the connectivity of the whole free space. Afterward, the second phase starts by specifying the initial and goal configurations, q_{init} and q_{goal}, respectively. Then the planner connects the two configurations to the roadmap, which is searched for a feasible path between them. The second type of sampling-based planners, the single-query planners, are based on the idea of growing a tree from q_{init} that will incrementally cover the free space until, eventually, it reaches q_{goal}. In 1990, Barraquand and Latombe introduced the planner that was later called the *Randomized Path Planner* (RPP), [2]. This planner was the first well known sampling-based planner. The *Ariadne's Clew* algorithm is another randomized sampling-based algorithm, [10]. It grows a search tree that is directed toward exploring new portions of the free space in each iteration. The main drawback of this algorithm is the difficult heuristic choices required for the highly parallelized genetic algorithm used during the exploration of C_{free}. The Expansive space planner is a single-query path planner which was introduced by Hsu, [4]. It tries to "push" the tree toward the unexplored area of the C-space. One limitation of this approach is that it requires substantial parameter tuning. Recently, *probabilistic roadmaps* became the most popular paradigm for sampling-based motion planning, [6]. The original probabilistic roadmap (PRM) follows the typical approach of the multiple-query planners. PRMs experienced major success in a large number of applications. However, PRMs failed to perform efficiently in motion planning problems that involve holonomic and nonholonomic constraints. Lately, the *Rapidly-Exploring Random Trees* (RRTs) were

Both authors are with the Department of Computer Science, University of Sharjah, P O Box 27272, Sharjah, UAE, **ashraf@sharjah.ac.ae**

introduced to remedy this shortage, [7].

The pursuit-evasion problem is considerably more complex than just the basic motion planning problem. Its first appearance was in pursuit-evasion games, [5]. Suzuki and Yamashita were the first to address the problem of pursuit-evasion in a polygonal environment, [12]. The first complete algorithm for a pursuer with omni-directional visibility was presented in [8]. All of these algorithms work in simply-connected planar environments that require explicit representation of the C-space. This implies that problems with high dofs or holonomic and differential constraints can not be handled. Recently Sampling Techniques for tracking unpredictable evaders have been studied in [13].

We present a practical algorithm that can be extended to higher dofs and take into account the holonomic and differential constraints. Further, we employ the Kd-Tree to efficiently query the vertices of the RRT for tracking purposes. The algorithm proves to be fast, simple and practical.

Section II describes the basic data structures (RRT and Kd-Tree) used by our algorithm. Details of the proposed algorithm are described in section III. The complexity analysis and completeness of the algorithm are discussed in section IV followed by simulation results in Section V. Finally, we conclude the work and present future research directions in Section VI.

II. DATA STRUCTURES

The Rapidly-Exploring Random Tree (RRT) is one of the latest and most successful randomized sampling-based algorithms. It was originally introduced by LaValle in 1998, [7]. Its success is due to its applicability to a wide range of motion planning problems which involve holonomic, nonholonomic and kinodynamic constraints. It is a single-query algorithm that starts by initializing a tree rooted at the starting configuration (q_{init}). RRTs incrementally search the C-space for a path connecting q_{init} and the goal (q_{goal}). At each iteration three main steps are executed: 1) new configuration is sampled at random (q_{rand}); 2) the nearest vertex from the current tree to this sample is computed (q_{near}); 3) and the tree is extended toward the sample point (add a new edge, with edge length = Δq, and a new vertex, q_{new}, to the tree in the direction of the sample point). The probability to choose a vertex for expansion is proportional to the area of its Voronoi region. Thus, RRTs are Voronoi-biased, [7].

We use a *Bi-directional* RRT to replace collision, whenever encountered. It is constructed by initializing two RRT trees rooted at two different configurations. At each iteration, one tree is extended and it attempts to connect the newly added vertex toward its nearest neighbor from the other tree. It should be noted that in the absence of motion constraints, the planner attempts a complete connection from q_{near} to q_{rand}, if possible. However, in the presence of holonomic, nonholonomic, and kinodynamic constraints, the planner adds a vertex, q_{new}, along the path from q_{near} to q_{rand}.

One bottleneck in RRT implementation is finding the nearest neighbor. This step has to be performed in each iteration. Naive nearest neighbor computations yield a linear-time algorithm. As a consequence, executing the RRT algorithm for M iterations with naive nearest neighbor computation will result in a quadratic runtime. Atramentov and La Valle have addressed a solution for this problem in [11].

After constructing the RRT, we use a Kd-Tree as a second data structure for storing the RRT vertices in order to yield a better nearest neighbor querying time. Kd-Tree is a powerful data structure for nearest neighbor problems, which was originally developed for database querying purposes.

The basic idea for building a Kd-Tree for a set of points (P) in the plane starts by sorting P (say along the x-axis). At the root, the set P will be split into two subsets P_1 and P_2 with a vertical line l_1. P_1 will contain the set of points on or to the left of l_1, and P_2 will contain the set of points to its right. The splitting line l_1 will be stored at the root, the subset P_1 will be stored at the left subtree of the root and the subset P_2 at the right subtree. P_1 will be again split, but with a horizontal line, after sorting the points along the y-axis. The subset of points below and on the line will be stored in the left subtree of P_1, and the subset of points above it will be stored in its right subtree. The splitting line will be stored in the left child node. Similarly, the set of points in the right subtree will be split with a horizontal line and the same division process will be applied. In general, at nodes of even depth we split with a vertical line, and at nodes with odd depth we split with a horizontal line. This recursive procedure will be applied repeatedly until we reach a subtree with a single point. This point will be stored as a leaf child of the last splitting line in this subtree. The algorithm **BuildKD_Tree()** describes the construction a 2d-tree:

Algorithm *BuildKD_Tree*(P,depth)
Input: A set of points, P, and the current depth, $depth$.
Output: The root of a kd-tree storing P.

1. **if** P contains only one point
2. **then return** a leaf storing this point
3. **else if** $depth$ is even
4. **then** Split P into two subsets with a vertical line l through the median x-coordinate of the points in P. Let P_1 be the set of points to the left of l or on l, and let P_2 be the set of points to the right of l
5. **else** Split P into two subsets with a horizontal line l through the median y-coordinate of the points in P. Let P_1 be the set of points below l or on l, and let P_2 be the set of points above l
6. $v_{left} \leftarrow$ BuildKD_Tree(P_1, $depth$+1)
7. $v_{right} \leftarrow$ BuildKD_Tree(P_2, $depth$+1)
8. Create a node v storing l, make v_{left} the left child of v, and make v_{right} the right child of v
9. **return** v

The algorithm **BuildKD_Tree()** requires two parameters: the set of points (P) and the current depth ($depth$), which determines the splitting direction. As a result from the Kd-Tree construction phase, a node v which is contained in the

tree will be included in a region that is bounded on one or more directions by splitting lines stored at its ancestors. We will refer to this region as $region(v)$. For instance, the region of the root is the entire plane. After the Kd-Tree is constructed, we proceed to the query phase. The query algorithm, **SearchKD_Tree()** , requires two parameters, the first is the root of the Kd-Tree, v, and the second is the query range, R, which is a rectangular area (if we are querying a 2-d Kd-Tree). The following is the query algorithm for the 2-d Kd-Tree:

Algorithm *SearchKD_Tree*(v,R)
Input: The root of (a subtree of) a Kd-Tree and a range, R.
Output: All points at leaves below v that lie in the range.

```
1.  if v is a leaf
2.    then Report the point stored at v if it lies in R
3.    else  if region(lc(v)) is fully contained in R
4.            then ReportSubTree(lc(v))
5.            else  if region(lc(v)) intersects R
6.                    then SearchKD_Tree(lc(v), R)
7.          if region(rc(v)) is fully contained in R
8.            then ReportSubTree(rc(v))
9.            else  if region(rc(v)) intersects R
10.                   then SearchKD_Tree(rc(v), R)
```

The main test in this algorithm is finding if the range R intersects the $region(v)$. The region of every node in the Kd-Tree can be computed and stored in v during the building process. The algorithm uses a subroutine **ReportSubTree**(v) to traverse the subtree rooted at v and report all the points stored in its leaves.

III. Evader(s) Detection & Tracking

We employ the RRT structure to detect evaders. In this work we assume that the evader will not move until it is spotted by a pursuer. The root of the tree, q_{init}, is the initial position of the pursuer. From this position, branches are added in all directions in the environment. The expansion process continues until the position of the evader is reached from one of the tree vertices, if possible, or the maximum number of iterations is reached. The user defined value, M, allows an estimate of the coverage percentile of the free space from the user perspective. This is a key parameter that controls the termination of the algorithm. As a result, if M is chosen smaller than what is required then the pursuer may not find the evader.

We generalize this idea to account for multiple pursuers searching for several evaders. In this case, the algorithm starts by initializing a set of trees ($T = \{T_1, T_2, ..., T_k\}$), where k is the number of pursuers, based on the locations given in Q_{init} ($\{q_1, q_2, ..., q_k\}$). Next, the nearest vertex, from all the existing RRTs, to the generated random configuration, q_{rand}, is determined. The algorithm and its analysis are described in [1].

A. Evader(s) Tracking

To account for tracking after detection, we continue building all RRTs until the free space is covered (i.e., M iterations), since tracking requires good and uniform coverage for the environment.

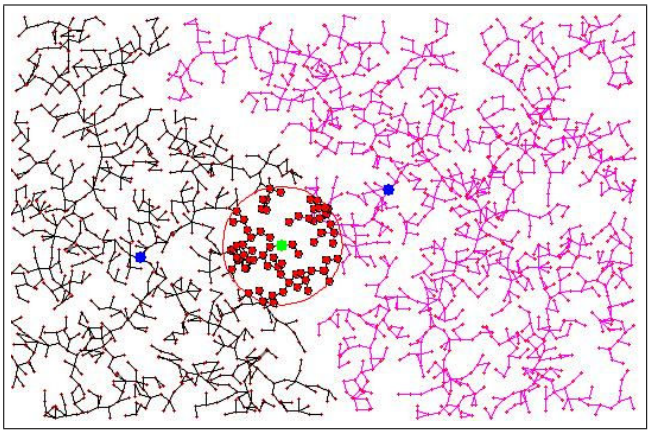

Fig. 1. An evader located on the borders of two RRTs and can be tracked by both pursuers in the environment.

The pursuers in our system are assumed to have omni-directional and bounded visibility (i.e., the pursuer can see the evader if it lies within a distance that is $\leq r$) where r is the radius of the evader's visibility field (a circle centered at the evader position). All the RRTs' vertices that are contained in this circle form potential pursuer future positions in order to the maintain visibility of the evader. The visibility field and its set of configurations are denoted by $VisCircle$ and $VisVer$, respectively. Figure (1) depicts the visibility field and visible configurations (in bold) of a pursuer.

Upon completion of the RRTs construction, all the vertices of the trees are sent to the **BuildKD_Tree()** procedure in order to generate the Kd-Tree, denoted by, T. Figure (2) shows an RRT that was constructed using 50 iterations and its corresponding Kd-Tree.

The algorithm **EvaderTracking()** takes as input the Kd-Tree, T, the evader to be tracked, evd, and the responsible pursuer, pur. Assigning pursuers to evaders is previously accomplished in the evaders detection phase.

Algorithm *EvaderTracking*(T,evd,pur)
Input: The Kd-Tree T, the evader to be tracked evd, and the pursuer tracking it, pur.
Output: Set of movements to maintain visibility of evd.

```
1.  VisVer = ComputeVisVer(T, evd);
2.  repeat
3.        if pur ∉ VisVer
4.          then v = ClosestVer(VisVer, pur);
5.               Move pur to v;
6.        VisVer = UpdateVisVer(T, evd);
7.  until tracking flag is off;
```

The algorithm uses two basic functions, which are described next.

Algorithm *ComputeVisVer*(T,evd)
Input: The Kd-Tree T and the evader to be tracked, evd.
Output: The list of visibile vertices $VisVer$.

1. R = EnclosingRectangle(evd, r);
2. $VisVer$ = SearchKD_Tree(v, R);
3. **for** each $v \in VisVer$
4. **if** (evd is invisibile to v)**or** (distance(evd,v)$> r$)
5. **then** remove v from $VisVer$;
6. **return** $VisVer$;

EnclosingRectangle() is used to retrieve a region representing the enclosing rectangle of $VisCircle$ in order to be used in the **SearchKD_Tree()** algorithm.

Algorithm *UpdateVisVer*(T,evd)
Input: The Kd-Tree T and the evader to be tracked evd.
Output: The list of visibile vertices $VisVer$.

1. **for** each $v \in VisVer$
2. **if** (evd is invisibile to v)**or** (distance(evd,v)$> r$)
3. **then** remove v from $VisVer$;
4. **if** $VisVer$ is empty
5. **then** $VisVer$ = ComputeVisVer(T, evd);
6. **return** $VisVer$;

Notice that in both, **ComputeVisVer()** and **UpdateVisVer()** , the pur does not move as long as it belongs to $VisVer$. If the evd moves, **UpdateVisVer()** "filters" the current $VisVer$ list and a new computation of VisVer is not required until it becomes empty. This contributes to a better time complexity.

Tracking multiple evaders poses few concerns that need to be addressed. Among which is how a pursuer would react if two evaders lie within its RRT boundary? Which one to track? Another concern is whether communication among pursuers is allowed.

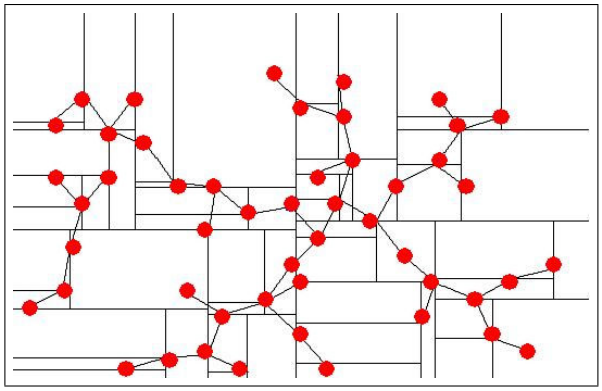

Fig. 2. An example of an RRT and its corresponding Kd-Tree

Each pursuer has an RRT associated with it, which covers a certain territory (denoted by $territory(pur)$). In the case of multiple evaders, one pursuer can have two, or more, evaders "wondering" around in its territory. Which evader should the pursuer track? We resolve this conflict by forcing the pursuer, pur, to follow the same evader it was assigned. The tracking will continue until the evader exits $territory(pur)$. Afterward, pur tracks the closest evader in $territory(pur)$, if any. However, the system allows for the closest evader to be always tracked by the closest pursuer. This is done for simulation purposes.

In the case of multiple pursuers, imagine the following setup: two pursuers, pur_1 and pur_2, and an evader, evd, lie in both of the pursuers' territories (i.e., $evd \in territory(pur_1)$ and $evd \in territory(pur_2)$). See Figure (1) for an example. If no communication exists between the pursuers both will track this evader. Otherwise, evd will be tracked by the pursuer it was assigned to until it exits its territory. Our system can simulate both scenarios.

B. Obstacle Avoidance and Collision Detection

The main idea in sampling-based motion planning algorithms is to avoid the explicit construction of the C_{obs}. Therefore, every sample and every edge between any two vertices along the solution path have to be checked for collision before adding it successfully to the resulting path. A number of collision detection approaches exist such as incremental and lazy detection. Sampling-based motion planning algorithms are always developed without taking collision detection into consideration.

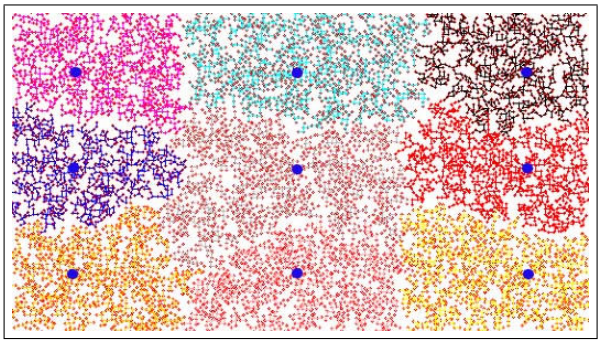

Fig. 3. An example showing the RRT s of 9 pursuers distributed evenly in a 2D environment. The RRTs cover the free space uniformly.

The *lazy* collision detection approach has proven to be the most effective detection module so far, [3]. A number of factors have led to the development of this module. Most important is that in sampling-based roadmaps, a large number of the roadmap edges are not on the final path to the goal, and as a result a good number of unneeded collision checks are performed. Lazy collision detection delays the collision checks until the edge is considered for traversal. If it is collision free it is added otherwise it is resolved. It has been noticed that most collisions occur at the corners of the obstacles. The remedy is to construct an alternative path by building small bidirectional RRTs rooted at the two ends of the invalid edge. Once a path is found, the vertices along the path are inserted in the Kd-Tree and all the remaining vertices will be "pruned". For RRTs that use a small value for the growth-rate (the edge length, Δq), collision checks are significantly reduced, but on the expense of more nearest neighbor queries.

TABLE I

SUMMARY DATA OF THE 9 PURSUERS IN FIGURE (3).

pursuer#	pursuers' Details		
	q_{init}	RRT size	Coverage %
1	(x: 490,y: 59)	621	12.42%
2	(x: 271,y: 59)	675	13.5%
3	(x: 60,y: 59)	515	10.3%
4	(x: 59,y: 249)	555	11.1%
5	(x: 271,y: 249)	678	13.56%
6	(x: 490,y: 250)	642	12.84%
7	(x: 490,y: 150)	332	6.64%
8	(x: 271,y: 151)	587	11.74%
9	(x: 60,y: 149)	395	7.9%

IV. COMPLEXITY ANALYSIS AND ALGORITHM COMPLETENESS

The nearest neighbor operations is the most costly one in the detection phase. A naive nearest neighbor results in $O(n^2)$ running time. The Kd-Tree construction, which follows the detection phase, is done only once. The most expensive step in the **BuildKD_Tree()** algorithm is finding the split line. But, since we sort the set of points, the cost of finding the median becomes constant. Therefore, sorting becomes the most expansive step, and the best sorting algorithms (*e.g.*, Merge Sort) takes $O(n\log n)$ time which is the total cost of constructing the Kd-Tree. In addition, it requires $O(n)$ storage where n is the number of points in P. On the other hand, the most expensive step in the tracking phase is querying the Kd-Tree. It takes $O(\sqrt{n}+ k)$ for every query using **SearchKD_Tree()** algorithm. The constant k represents the number of vertices retrieved from the query. Therefore, the algorithm is output sensitive. The querying is performed every time an evader moves out of the previously computed visibility field using the two subroutines **ComputeVisVer()** and **UpdateVisVer()** . Thus, the number of queries needed at any instance of time is equal to the number of evaders moved out of visibility field multiplied by the cost of a single query. The complexity of the subroutine **ReportSubTree()** , used by the algorithm **SearchKD_Tree()**, is linear in the number of vertices reported.

All sampling-based algorithms experience a form of completeness called the *probabilistic completeness*. It means, the probability of finding collision-free path approaches one as time goes to infinity. For our algorithm, the probability that the RRT rooted at q_{init} will contain q_{goal} as a vertex approaches 1 as the number of vertices in the RRT approaches ∞. Formally, it is $\lim_{n\to\infty} P[q_{goal} \in RRT(q_{init})] = 1$.

V. EXPERIMENTAL RESULTS

This section presents a complete set of simulations carried out on an 800 MHz Pentium III PC. The input parameter Δq (edge length) is 10 and the maximum number of iterations M is 2000. RRT building process is not stopped when all evaders are detected, rather it continued until M is reached in order to ensure uniform and even coverage of the

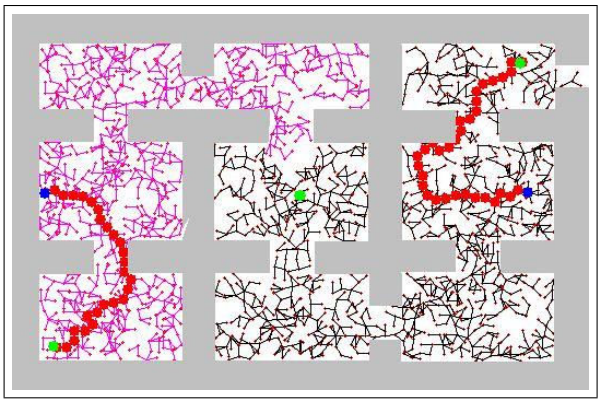

Fig. 4. The simulation after the detection phase. The paths of the two pursuers are shown in bold.

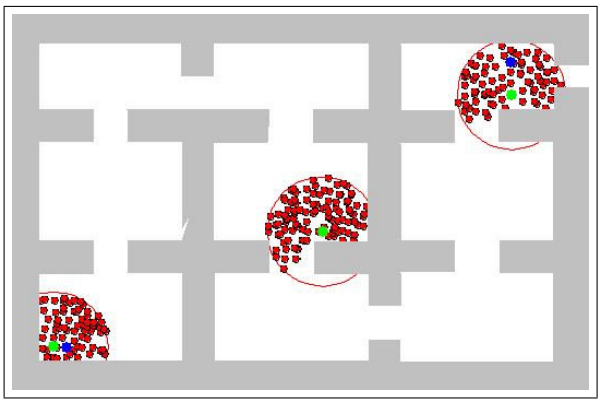

Fig. 5. Tracking snap shot at t_0.

environment. While the pursuer(s) configuration(s) (q_{init}) is shown as a small solid circle(s) in dark color, the evaders are shown in light color. Notice that we enlarged the size of each configuration point (i.e., solid circle with radius of 5 pixels) for clarity of presentation.

Figure (3) shows the resulting RRTs for the environment in which nine pursuers were evenly distributed. Δq is set to 5 distance-units. The computed set of RRTs cover the environment uniformly. The building process was accomplished in 17.5 seconds. Table (I) includes information about each pursuer. Namely, each pursuer starting configuration, the size of its RRT (number of nodes), and the coverage ratio on the environment. Although RRTs are constructed at random, they (more or less) maintain a uniform coverage of the whole environment. Heuristics may be introduced to enforce balanced distributions among pursuers. However this depends on the application it is used for. The simulation was carried out for the environment in Figure (4). It contains two pursuers and three evaders distributed as shown in the figure along with the resulting RRTs. The left pursuer detected the left lower evader after 0.21 seconds with 435 iterations while the right pursuer detected the other two. The upper right evader was detected in 0.38 seconds (540 iterations) while the middle evader was found in 1.2 seconds (1058 iterations). The difference in time (4.6 the total time and 1.2 the time of detecting the last evader) is due to the

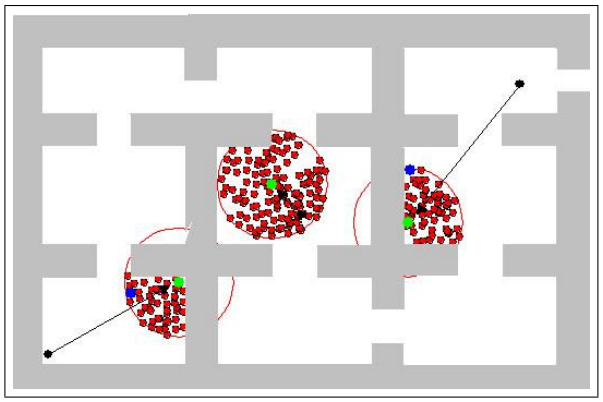

Fig. 6. Tracking snap shots after Δt_1.

completion of the building process until reaching M. One can notice the uniform and even coverage of the environment. The left pursuer's RRT contains 867 vertices, which results in a coverage ratio of about 43.35%. On the other hand, the right pursuer covers 56.65% from the area explored by the RRTs using 1133 vertices in its RRT. Although RRTs are constructed at random, they (more or less) maintain a uniform coverage of the whole environment.

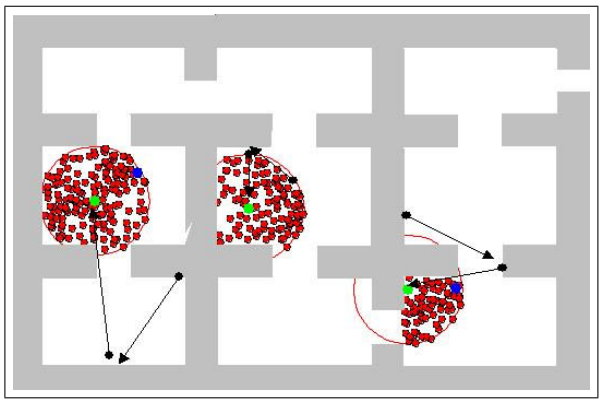

Fig. 7. Tracking snap shots after Δt_2.

In Figure (4), where the free paths get very close to an obstacle boundary; it might lead to thinking of a collision with that obstacle. It is not the case. Although, the right pursuer detected two evaders, the path to the closest one is only shown, since in the tracking phase the pursuer is only capable of pursuing one evader.

Figures (5) to (7) depicts an example of tracking three evaders by two pursuers. The first Figure shows the tracking at t_0 where the pursuers have moved forward to the evaders they have detected (for the right pursuer, it moved to the closer evader it detected). The $VisCircle$s along with their contents are displayed around each evader. The empty parts of some of the $VisCircles$ may lead the viewer to think that this area does not contain RRT vertices. But the fact that this is due to the removal of the $VisVer$ configurations (RRT vertices)that are not "seen" by the evader because of the existence of one or more obstacles between this vertex and the evader. The two other Figures show the environment after Δt_1 and Δt_2 times, respectively. The movements (distances and directions) of the evaders are shown by the black arrows. The base of the arrow represents the evaders position before the movement, which is also the last position it reached at the last Δt time. All evaders and pursuers have the same speed. The evaders moved at *random* linearly, which caused the evader to move in a straight-line until an obstacle is met, followed by random turns until the evader is able to move forward again without collision.

VI. CONCLUSIONS AND FUTURE WORKS

We proposed the use of RRT in pursuit-evasion problems. We used a modified RRT algorithm to allow multiple pursuers and evaders. The checking for the evader(s) is done during the construction phase of the RRT(s). The detection was followed by a tracking phase where the Kd-Tree is used to store the RRT vertices in order to cut down the search space when an RRT vertex is selected by the pursuer in order to maintain visibility of the evader. To improve the performance of the algorithm, a lazy collision detector is utilized for obstacle avoidance at runtime. In general, RRTs have proved to perform faster than the basic probabilistic roadmaps for holonomic, nonholonomic and kinodynamic motion planning problems. It promises the motion planning field with a brighter future.

REFERENCES

[1] A. Elnagar and A. Aldahak. RRT-Based Multiple Evaders Detection algorithm. In *Proceedings of the International Conference on Automation, Robotics and Autonomous Systems*, pages 44–50, 2005.

[2] J. Barraquand and J. C. Latombe. A monte-carlo algorithm for path planning with many degrees of freedom. *IEEE International Conference on Robotics and Automation*, pages 1712–1717, 1990.

[3] R. Bohlin and L. E. Kavraki. Path planning using lazy prm. *Proceedings of the IEEE International Conference on Robotics and Automation*, 1:521–528, April 2000.

[4] D. Hsu, J. C. Latombe, and R. Motwani. Path planning in expansize configuration spaces. *International Journal of Computational Geometry and Applications*, 4:495–512, 1999.

[5] R. Isaacs. *Differential Games*. Wiley, New York, NY, 1965.

[6] L. E. Kavraki, P. Svestka, J. C. Latombe, and M. Overmars. Probabilistic roadmaps for fast path planning in high dimensional spaces. *IEEE Transactions on Robotics and Automation*, 12:566–580, 1996.

[7] S. M. LaValle. Rapidy-exploring random trees: A new tool for path planning. *TR98-11*, October 1998.

[8] S. M. LaValle, D. Lin, L. J. Guibas, J. C. Latombe, and R. Motwani. Finding an unpredictable target in a workspace with obstacles. *Proceedings of the International Conference on Robotics and Automation*, pages 737–742, 1997.

[9] T. Lozano-Perez and M. A. Wesley. An algorithm for planning collision-free paths among polyhedral obstacles. *Communication of the ACM*, 22(10):560–570, 1979.

[10] E. Mazer, J. M. Ahuactzin, and P. Bessiere. The ariadne's clew algorithm. *Journal of Artificial Intelligence Research*, 9:295–316, November 1998.

[11] A. Atramentov, and S. M. La Valle. Efficient Nearest Neighbor Searching for Motion Planning. *In Proceedings IEEE International Conference on Robotics and Automation*, pages 632–637, 2002.

[12] I. Suzuki and M. Yamashita. Searching for a mobile intruder in a polygonal region. *SIAMJ. Comput.*, 21(5):863–888, 1992.

[13] R. Murrieta-Cid, B. Tovar and S. Hutchinson. A Sampling-based Motion Planning Approach to Maintain Visibility of Unpredictable Targets. *Journal on Autonomous Robots*, 19(3):285–300, December 2005.