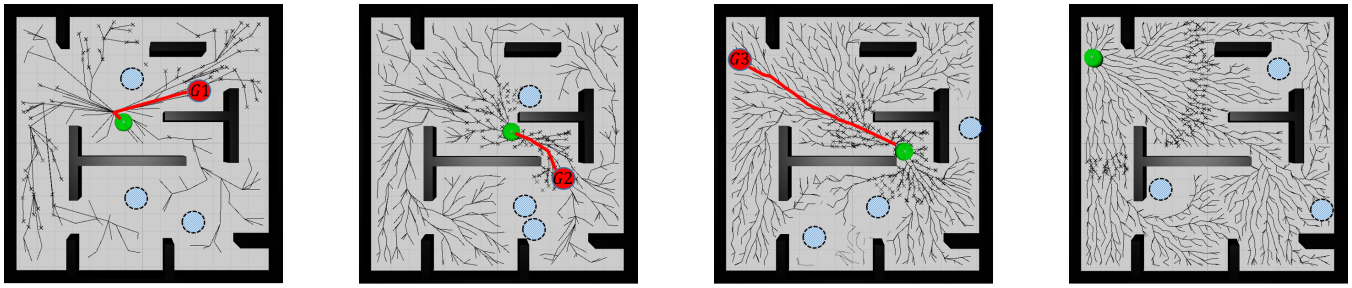
RT-RRT*: A Real-Time Path Planning Algorithm Based On RRT*

Kourosh Naderi* Joose Rajamäki† Perttu Hämäläinen‡

Aalto University



(a) (b) (c) (d)

Figure 1: *Bold lines and shaded circles denote paths from the agent to different goals and dynamic obstacles, respectively. In Figure (a) the tree is growing and a path to the goal point is found. In Figures (b) and (c) the goal point is changed on the fly. In both cases since the tree root moves with the agent and the tree covers most of the environment, the paths to the changed goals are returned quickly. In Figure (d) the agent has reached the goal point and rewiring of the nodes based on the current location of the tree root has generated the minimum length paths to others nodes of the tree. The crossed points denote the Rewiring Circle.*

Abstract

This paper presents a novel algorithm for real-time path-planning in a dynamic environment such as a computer game. We utilize a real-time sampling approach based on the Rapidly Exploring Random Tree (RRT) algorithm that has enjoyed wide success in robotics. More specifically, our algorithm is based on the RRT* and informed RRT* variants. We contribute by introducing an online tree rewiring strategy that allows the tree root to move with the agent without discarding previously sampled paths. Our method also does not have to wait for the tree to be fully built, as tree expansion and taking actions are interleaved. To our knowledge, this is the first real-time variant of RRT*.

We demonstrate our method, dubbed Real-Time RRT* (RT-RRT*), in navigating a maze with moving enemies that the controlled agent is required to avoid within a predefined radius. Our method finds paths to new targets considerably faster when compared to CL-RRT, a previously proposed real-time RRT variant.

CR Categories: I.2.8 [Computing methodologies]: Artificial Intelligence—Problem Solving, Control Methods, and Search I.2.9 [Computing methodologies]: Artificial Intelligence—Robotics

Keywords: Real-time path planning, RRT*, dynamic environment, multiple-query planning

*e-mail:kourosh.naderi@aalto.fi
†e-mail:joose.rajamaki@aalto.fi
‡e-mail:perttu.hamalainen@aalto.fi

MIG '15, November 16 – 18, 2015, Paris, France.
 ISBN 978-1-4503-3991-9/15/11…$15.00
DOI: http://dx.doi.org/10.1145/2822013.2822036

1 Introduction

Path planning in dynamic environments is a demanding problem encountered in many robotic tasks and computer games [Rastgoo et al. 2014; Sud et al. 2008]. Real-time path planning algorithms are used to react to the changes in the environment as well as to constantly look for a better path to the goal point. These algorithms have to balance with the trade-off of the goodness of the path versus having a short search time. The path planning problem gets even more challenging when changing the goal point is allowed, which is the case in many multiple-query tasks [Kavraki et al. 1996], e.g. computer games. Difficulty in path planning arises also when the structure of the agent gets more sophisticated and controlling the agent is not trivial [Sud et al. 2008], e.g. planning a path for a humanoid agent [Gutmann et al. 2005].

Many of the current approaches for solving the real-time path planning problem fall into the categories of heuristic methods [Gutmann et al. 2005], potential field methods [Khatib 1986] and sampling methods like rapidly exploring random trees (RRTs) [LaValle 1998]. A survey of approaches to real-time path planning in dynamic environment can be found in [Rastgoo et al. 2014]. Most of the algorithms used in the games industry stem from the A* algorithm. The A* algorithm, however, faces prolonged search time in an environment that has many complex obstacles. The real-time versions of A* algorithm inherit this problem and try to solve it using various methods [Cannon et al. 2012; Sturtevant et al. 2010]. Furthermore, A* needs discretization of the environment whose resolution has a strong effect on the search time. Although in [Cannon et al. 2012] the algorithm samples for building a graph, the number of samples for finding a path between nodes increases such that it may affect the real-time response time of the algorithm for taking an action. The response time of the potential field algorithms is in real-time, however they suffer from trapping into local minima.

Rapidly exploring random trees (RRTs) quickly expand a search tree in an environment. They are efficient for single-query tasks in a continuous environment, i.e. when the goal point is fixed. The different real time variants of RRT path planning either regrow the whole tree with guidance of the previous iteration for a limited time

[Bruce and Veloso 2002] or prune infeasible branches of the tree after changing its root [Luders et al. 2010]. Unlike these algorithms, our proposed algorithm retains the whole tree in the environment and rewires the nodes of the tree based on the location of the tree root and changes in the dynamic obstacles. Consequently, our algorithm enables the user to rapidly switch the goal point around (multiple-query task) by efficiently using the expanded tree in the environment (see Fig 1). To the best of our knowledge, our algorithm is the first real-time RRT-based algorithm that retains the whole tree, gradually rewires different parts of the tree, and works efficiently in multiple query tasks.

In the following, we first provide a brief review of the relevant literature on real-time path planning algorithms in Section 2. Section 3 formally defines the real-time path planning problem and introduces the related notation. A description of our proposed RT-RRT* method and our simulation experiments then follow in Sections 4 and 5. The simulations indicate that our method outperforms CL-RRT which can be considered the state-of-the-art of real-time RRT-based path planning methods.

2 Related Work

In this section we first review the relevant literature regarding tree based algorithms for path planning and in the latter subsection we introduce the currently used approaches to real-time path planning.

2.1 RRTs and their extensions

RRTs sample points in the space and add them to a tree which grows to the whole planning space. RRTs [LaValle 1998] have the useful properties of: 1) covering the whole space efficiently and quickly; 2) probabilistic completeness i.e. as the number of nodes in the tree increases, the probability of finding a solution approaches one. However, RRTs are not asymptotically optimal and rewiring is not performed in the RRT algorithm, i.e. the connections between nodes are set once. The emergence of RRT* [Karaman and Frazzoli 2011] changed this as it allows to rewire the tree connections such that the path length from the root to a leaf is reduced. RRT* is asymptotically optimal, but its convergence is slow especially in large environments. Informed RRT* [Gammell et al. 2014] introduced a focused sampling method that samples new nodes inside an ellipsoid. The focal points of the ellipsoid are the starting and goal points. This method increased the convergence rate of RRT* especially in large environments. In this paper, we combine the good aspects of RRT variants and we introduce real-time path planning that is based on RRT* and Informed RRT*.

2.2 Real-time Path Planning methods

Tree Based Path Planning methods mostly stem from RRTs. Two of these methods are ERRT [Bruce and Veloso 2002] and CL-RRT [Luders et al. 2010]. The tree of these algorithms covers a small portion of the environment. Therefore, they only use the tree as a look-ahead in their path planning, which reduces the search time but increases the length of the path to the goal. At each iteration ERRT creates a tree using some way-points of the previous iteration. CL-RRT on the other hand ensures that the agent will not deviate from the planned path and prunes infeasible branches when the agent moves on the tree. Contrary to these methods, we retain the tree between the iterations and change the tree root when the agent moves. Also, we rewire the tree when the tree root changes or a dynamic obstacle blocks a node. As a result, our method needs very few iterations to search for a path to various goal points as the tree grows, and the paths to those goal points are shorter compared to CL-RRT (see Section 5). This also makes our method suitable for multi-query tasks, i.e. querying paths to multiple goals, which is highly preferable algorithm quality e.g. in games. The first real-time motion re-planning on RRT* is RRT$^{\text{X}}$ [Otte and Frazzoli 2015]. As opposed to RRT$^{\text{X}}$, our method is designed for multi-query tasks, which plans paths from agent to goals, and interleaves tree expansion and rewiring with taking actions.

Potential Based Path Planning methods treat the environment as a potential field such that the goal point attracts and the obstacles repulse the agent. These methods stem from the original Artificial Potential Field (APF) introduced in [Khatib 1986]. In spite of being useful as a real-time path planning, these methods suffer from trapping into local minima. Thus, potential field methods need additional effort to overcome the problem of local minima as well as to find a minimum length path to the goal point.

Graph Based Path Planning methods usually make a grid of the environment and apply real-time versions of A* to it. Some of the methods simply divide the environment into simple polygonal grids [Sturtevant et al. 2010; Gutmann et al. 2005]. On the other hand, there are some methods that use Voronoi diagram [Sud et al. 2008] or sampling [Cannon et al. 2012] to build a graph representing the environment. One disadvantage of graph based over tree based path planning is that even though the environment is explored and a graph representing it is constructed, one needs further processing, such as A*, to extract the path from the graph. In multi-query tasks, graph based path planning methods may need to search the entire graph to find a path to different goal points. Unlike this, our proposed algorithm needs fewer iterations for finding goal points as the tree grows in the environment. Besides, RT-RRT* utilizes the efficient tree structure to return a path to possibly multiple goal points by backtracking the ancestors from the goal.

3 Problem Statement and Notations

Let us denote the work space, where path planning is done, by $\mathcal{X}$. In our algorithm $\mathcal{X}$ can be subset of $\mathbb{R}^2$ or $\mathbb{R}^3$. However, without loss of generality in this paper we only consider $\mathcal{X} \subseteq \mathbb{R}^2$. Also, $\mathcal{X}$ is considered to be bounded. Besides spaces such as $\mathcal{X}$, the calligraphy alphabet is used to refer to sets. $\mathcal{X}$ contains obstacles, which may be dynamic. $\mathcal{X}_{\text{obs}} \subsetneq \mathcal{X}$ denotes the set of all obstacles in the environment, and we assume it is known. Thus, the free space is denoted by $\mathcal{X}_{\text{free}} = \mathcal{X} \setminus \mathcal{X}_{\text{obs}}$. The tree is denoted by $\mathcal{T}$ and each node inside it is represented by $\mathbf{x}_{\text{i}} \in \mathcal{X}_{\text{free}}$. The current tree root is denoted by $\mathbf{x}_0$ which is changed when the agent moves. In the path planning algorithm the set of planned nodes is represented by $(\mathbf{x}_0, \mathbf{x}_1, ..., \mathbf{x}_k)$ in which k is the user-defined limit for planning a path ahead from $\mathbf{x}_0$. Besides nodes, $\mathbf{x}$ is used in our algorithm to denote any positions in the environment. For example, $\mathbf{x}_{\text{goal}} \in \mathcal{X}_{\text{free}}$ represents the goal point, which can be changed by the user. $\mathbf{x}_{\text{a}} \in \mathcal{X}_{\text{free}}$ is the position of the agent.

We formulate the problem of real-time path planning in dynamic environments as follows. We want to find a path, i.e. ($\mathbf{x}_0$, $\mathbf{x}_1$, ..., $\mathbf{x}_{\text{goal}}$), from the agent to $\mathbf{x}_{\text{goal}}$ where $\mathbf{x}_{\text{goal}}$ can be changed on the fly (see Fig 1). Also, the path to $\mathbf{x}_{\text{goal}}$ should be of minimum length. For finding a minimal length path, we use cost-to-reach values (denoted by c_{i}) which are computed using the Euclidean length of the path from $\mathbf{x}_0$ to $\mathbf{x}_i$. Furthermore, we restrict our algorithm to be real-time. This means that we have a limited amount of time for Tree Expansion-and-Rewiring and Path Planning with the expanded tree. In real-time path planning it is important to have a real-time response whether or not we have a path to the goal. When the tree is growing and $\mathbf{x}_{\text{goal}}$ is not found, we use the cost function $f_{\text{i}} = c_{\text{i}} + h_{\text{i}}$ to plan a path from $\mathbf{x}_0$ to a point close to $\mathbf{x}_{\text{goal}}$. h_{i}

denotes cost-to-go values from $\mathbf{x}_i$ to $\mathbf{x}_{goal}$ and is computed using Euclidean distance between these two nodes. Note that, when time for path planning is up, next immediate node after $\mathbf{x}_0$ in the planned path, $\mathbf{x}_1$, is committed and should be followed.

In multi-query tasks, each point in $\mathcal{X}_{free}$ of the environment has the potential of being $\mathbf{x}_{goal}$ while dynamic obstacles are moving around and may block some paths. In our algorithm we change $\mathbf{x}_0$ when the agent moves to keep the agent near the tree root, and by retaining the whole tree between iterations we can return a path to any point in the environment (see Fig 1). Therefore, our main problem is, how to rewire the tree really fast and in a real-time manner to react to the changes in the environment (changes in $\mathcal{X}_{obs}$) as well as to return a minimal length path to $\mathbf{x}_{goal}$ while $\mathbf{x}_{goal}$ can be any point in the environment.

4 Method

Our method, which is introduced in Algorithm 1, interleaves path planning with tree expansion and rewiring. We initialize the tree with $\mathbf{x}_a$ as its root (line 2). At each iteration, we expand and rewire the tree for a limited user-defined time (lines 5-6). Then we plan a path from the current tree root for a limited user-defined amount of steps further (k in line 7). The planned path is a set of nodes starting from the tree root, $(\mathbf{x}_0, \mathbf{x}_1, ..., \mathbf{x}_k)$. At each iteration we move the agent for a limited time to keep it close to the tree root, $\mathbf{x}_0$ (line 10). When path planning is done and the agent is at the tree root, we change the tree root to the next immediate node after $\mathbf{x}_0$ in the planned path, $\mathbf{x}_1$ (lines 8-9). Hence, we enable the agent to move on the planned path on the tree towards the goal.

Algorithm 1 RT-RRT*: Our Real-Time Path Planning

1: **Input:** $\mathbf{x}_a$, $\mathcal{X}_{obs}$, $\mathbf{x}_{goal}$
2: Initialize $\mathcal{T}$ with $\mathbf{x}_a$, $\mathcal{Q}_r$, $\mathcal{Q}_s$
3: **loop**
4: `Update` $\mathbf{x}_{goal}$, $\mathbf{x}_a$, $\mathcal{X}_{free}$ and $\mathcal{X}_{obs}$
5: **while** time is left for `Expansion` and `Rewiring` **do**
6: `Expand` and `Rewire` $\mathcal{T}$ using Algorithm 2
7: Plan ($\mathbf{x}_0$, $\mathbf{x}_1$, ..., $\mathbf{x}_k$) to the goal using Algorithm 6
8: **if** $\mathbf{x}_a$ is close to $\mathbf{x}_0$ **then**
9: $\mathbf{x}_0 \leftarrow \mathbf{x}_1$
10: `Move` the agent toward $\mathbf{x}_0$ for a limited time
11: **end loop**

Algorithm 2 Tree Expansion-and-Rewiring

1: **Input:** $\mathcal{T}$, $\mathcal{Q}_r$, $\mathcal{Q}_s$, k_{max}, r_s
2: `Sample` $\mathbf{x}_{rand}$ using (1)
3: $\mathbf{x}_{closest} = \arg\min_{\mathbf{x} \in \mathcal{X}_{SI}}$ `dist`$(\mathbf{x}, \mathbf{x}_{rand})$
4: **if** `line`$(\mathbf{x}_{closest}, \mathbf{x}_{rand}) \subset \mathcal{X}_{free}$ **then**
5: $\mathcal{X}_{near}$ = `FindNodesNear`$(\mathbf{x}_{rand}, \mathcal{X}_{SI})$
6: **if** $|\mathcal{X}_{near}| < k_{max}$ or $|\mathbf{x}_{closest} - \mathbf{x}_{rand}| > r_s$ **then**
7: `AddNodeToTree`$(\mathcal{T}, \mathbf{x}_{rand}, \mathbf{x}_{closest}, \mathcal{X}_{near})$
8: `Push` $\mathbf{x}_{rand}$ to the first of $\mathcal{Q}_r$
9: **else**
10: `Push` $\mathbf{x}_{closest}$ to the first of $\mathcal{Q}_r$
11: `RewireRandomNode`$(\mathcal{Q}_r, \mathcal{T})$
12: `RewireFromRoot`$(\mathcal{Q}_s, \mathcal{T})$

$\mathcal{Q}_r$, $\mathcal{Q}_s$ are two different queues initialized in line 2 to be used for Rewiring in Algorithm 2. Line 4 of Algorithm 1 updates the goal point, position of the agent and the obstacles in the environment that is used later in the Path Planning and the Tree Expansion-and-Rewiring algorithms. We use control particle belief propagation algorithm (C-PBP) [Hämäläinen et al. 2015] in line 10 for moving the agent to separate path planning and synthesizing the motions of the agent, a practice bearing resemblance to [Song et al. 2014][Gutmann et al. 2005]. However, any other control algorithm can be used for moving the agent. Sections 4.1 and 4.2 explain the previously used Tree Expansion-and-Rewiring and Path Planning algorithms, respectively.

4.1 Tree Expansion and Rewiring

Tree Expansion-and-Rewiring is introduced in Algorithm 2. Sampled nodes, $\mathbf{x}_{rand}$ in line 2, are added to the tree until it completely covers the environment (line 7). The sampled node $\mathbf{x}_{rand}$ is always used in rewiring random parts of the tree either around itself or around its closest node, $\mathbf{x}_{closest}$ (lines 8, 10, 11). This is needed because of the changes in the tree root and dynamic obstacles. Fig 1d shows the situation when the tree has stopped adding nodes. Line 6 states the condition for this. k_{max} denotes maximum number of neighbors around a node such as $\mathbf{x}_{rand}$ and r_s is used for the maximum Euclidean distance allowed between the nodes in the tree. These two values together control the density of the tree. Same as with RRT*, $\mathcal{X}_{near}$ in line 5 is the set of nodes neighboring $\mathbf{x}_{rand}$. Line 4 checks whether the path between $\mathbf{x}_{rand}$ and $\mathbf{x}_{closest}$ is collision free or not. Lines 11 and 12 perform the two different rewiring methods in a large tree with a changing tree root. Their operation is explained in further depth in Section 4.1.2. What happens in line 7 is explored in Section 4.1.1.

Random sampling: Sampling in line 2 of Algorithm 2 uses the following equation. P_r is a random number between $[0, 1]$ and α is a small user-given constant, e.g. 0.1. $\beta \in \mathbb{R}$ is for dividing the sampling between Uniform$(\mathcal{X})$ and Ellipsis$(\mathbf{x}_0, \mathbf{x}_{goal})$ samplings.

$$\mathbf{x}_{rand} = \begin{cases} \text{LineTo}(\mathbf{x}_{goal}) & \text{if } P_r > 1 - \alpha \\ \text{Uniform}(\mathcal{X}) & \text{if } \begin{cases} P_r \leq \dfrac{1-\alpha}{\beta} & \text{or} \\ \nexists \text{path}(\mathbf{x}_0, \mathbf{x}_{goal}) \end{cases} \\ \text{Ellipsis}(\mathbf{x}_0, \mathbf{x}_{goal}) & \text{otherwise} \end{cases} \quad (1)$$

In (1), LineTo$(\mathbf{x}_{goal})$ samples randomly in the line between $\mathbf{x}_{goal}$ and the node of the tree that is closest to $\mathbf{x}_{goal}$. Uniform$(\mathcal{X})$ samples the environment uniformly. Finally, Ellipsis$(\mathbf{x}_0, \mathbf{x}_{goal})$ samples inside an ellipsis so that the path from $\mathbf{x}_0$ to $\mathbf{x}_{goal}$ is inside it. Same as Informed RRT* we need to sample the environment uniformly until we find a path to $\mathbf{x}_{goal}$. Due to the changes in the tree root and $\mathbf{x}_{goal}$ in our algorithm, we constantly need to rewire random parts of the tree and explore the environment for later queries in multi-query tasks. Thus, as opposed to Informed RRT*, when a path to $\mathbf{x}_{goal}$ is found, we focus part of the sampling inside the Ellipsis instead of all of it. Therefore, we can efficiently rewire the paths to $\mathbf{x}_{goal}$ and to the other parts of the tree as well. To sample in the Ellipsis, [Gammell et al. 2014] sets $\mathbf{x}_0$ and $\mathbf{x}_{goal}$ as focal points of the ellipsis where its transverse and conjugate diameters equal to c_{best} and $\sqrt{c_{best}^2 - c_{min}^2}$. c_{best} is the length of the path from $\mathbf{x}_0$ to $\mathbf{x}_{goal}$, and c_{min} is $\|\mathbf{x}_0 - \mathbf{x}_{goal}\|_2$. One should notice that unlike Informed RRT* we update the rotation of the ellipsis at every iteration because of the changing tree root and goal point.

Tree Largeness: By retaining $\mathcal{T}$ between iterations, the tree grows too large (but with a limited number of the nodes) to be handled wholly in real-time path planning. Instead, we use a subset of the nodes, denoted by $\mathcal{X}_{SI}$. For the sake of simplicity and saving memory we use grid-based spatial indexing (Fig 2 left), however one might use KD-Tree spatial indexing to gain even more speed up. $\mathcal{X}_{SI}$ is used to find $\mathbf{x}_{closest}$ and $\mathcal{X}_{near}$ of $\mathbf{x}_{rand}$ in Algorithm 2.

Also, $\mathcal{X}_{\text{SI}}$ is used in Algorithms 4 and 5 to search for $\mathcal{X}_{\text{near}}$ around $\mathbf{x}_{\text{r}}$ and $\mathbf{x}_{\text{s}}$, respectively. In addition, $\mathcal{X}_{\text{SI}}$ is used to search for a node inside the goal region (r_g) as well as to block every node inside the obstacle regions (r_b) of the dynamic obstacles that are within the distance of r_o from the agent (see Fig 2).

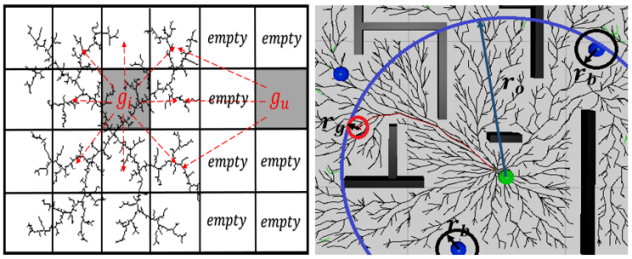


Figure 2: *Adjacent grid squares for g_u and for g_i (Left). Obstacle and goal regions are denoted by r_b and r_g, respectively (Right). Only effects of dynamic obstacles within the distance of r_o from the agent are considered.*

In order to work with $\mathcal{X}_{\text{SI}}$ instead of the whole $\mathcal{T}$, we divide the environment with a square grid (Fig 2 left). To find $\mathcal{X}_{\text{SI}}$ around one node (e.g. $\mathbf{x}_{\text{u}}$), we find the grid square g_{u}, where $\mathbf{x}_{\text{u}}$ is located. Then, we return all nodes of the tree that are inside g_{u} and in its adjacent squares. A grid square g_{j} is considered adjacent to other grid square g_{i} when g_{j} is the closest grid square to g_{i} which has at least one node of the tree inside it. Fig 2 illustrates the difference between adjacent grid squares for g_{i} and g_{u}. Note that, the size of the grid affects strongly the processing time and the size of $\mathcal{X}_{\text{SI}}$. Also, each grid should be big enough to contain the obstacle region around each dynamic obstacle (see Section 5 for our obstacle and grid details).

Blocking Nodes by Dynamic Obstacles: When a dynamic obstacle blocks nodes inside itself, we set their cost-to-reach values c_{i} to infinity. As a result, all of the nodes of the branches which stem from nodes in the obstacle region get infinity c_{i} value. Thus, when Rewiring algorithms are gradually and continuously connecting nodes with high c_{i} to their neighbors to reduce their c_{i}, rewiring will create another path for nodes with infinite c_{i}. Section 4.1.2 explains how rewiring the nodes in large trees is done. The computation of c_{i} is explained in Section 4.1.1.

Neighbor Radius: `FindNodesNear` in Algorithm 2, same as RRT*, returns the nodes in $\mathcal{X}_{\text{SI}}$ that are at most at distance of ε from $\mathbf{x}_{\text{rand}}$. We want to have control over the expected amount of nodes that `FindNodesNear` returns which should be k_{max} in Algorithm 2. We know that $k_{\text{max}} : n_{\text{total}}$ equals $\mu(B_\varepsilon) : \mu(\mathcal{X})$ where $A : B$ means ratio of A to B. n_{total} denotes total number of the nodes in the tree. $\mu(B_\varepsilon)$ is the volume/area of ε-radius ball, in 2D this would be $\pi\varepsilon^2$. $\mu(\mathcal{X})$ is the volume/area of the search space. Thus, ε is calculated as follows:

$$\varepsilon = \sqrt{\frac{\mu(\mathcal{X}) k_{max}}{\pi n_{\text{total}}}} \tag{2}$$

If ε in (2) becomes smaller than r_s in Algorithm 2, we set ε to r_s since r_s controls the closeness of the nodes. In Section 4.1.2 we use `FindNodesNear` to find $\mathcal{X}_{\text{near}}$ of different nodes, and gradually spread rewiring through the entire tree. Spreading the rewiring is directly related to the closeness of nodes and the selection of the neighbor nodes $\mathcal{X}_{\text{near}}$. Thus, setting ε smaller than r_s obstructs spreading the rewiring.

4.1.1 Adding a Node to the Tree

When we want to add $\mathbf{x}_{\text{new}}$ to the tree, same as with RRT*, Algorithm 3 finds the parent with the minimum cost-to-reach (c_{i}) value inside $\mathcal{X}_{\text{near}}$ (lines 2-6). One should note that the computation of c_{i} is related to the length of the path from $\mathbf{x}_0$ to $\mathbf{x}_{\text{i}}$. Thus, $\texttt{cost}(\mathbf{x}_{\text{i}})$ needs to compute c_{i} whenever c_{j} for any intermediate node in the path to $\mathbf{x}_{\text{i}}$ is changed. Therefore, when $\texttt{cost}(\mathbf{x}_{\text{i}})$ is called, it will recompute c_{i} from $\mathbf{x}_{\text{i}}$ up to $\mathbf{x}_0$ if a new node has been added to the path to $\mathbf{x}_{\text{i}}$ (same as RRT*) or any changes in $\mathbf{x}_0$ or dynamic obstacles inside r_o (see Fig 2) have happened (as opposed to RRT*). Note that whenever $\texttt{cost}(\mathbf{x}_{\text{i}})$ is called, if one of the ancestors of $\mathbf{x}_{\text{i}}$ is blocked by a dynamic obstacle, i.e. $c_{\text{i}} = \infty$, we block that node as well. The introduced $\texttt{cost}(\mathbf{x}_{\text{i}})$ is used in Algorithms 3-6.

$V_{\mathcal{T}}$ and $E_{\mathcal{T}}$ denote sets of nodes and edges in the tree, respectively. Note that, when a node is added to the tree, 1) it expands the tree; 2) if it is inside the goal region (Fig 2), a path to $\mathbf{x}_0$ is found (See Section 4.2); 3) we have to update the adjacent grid squares used for grid based spatial indexing (See Section 4.1). Also, it should be noted that we are using a tree structure to build the tree and each node has access to its children and its parent. In other words, we use $E_{\mathcal{T}}$ only in the explanation.

Algorithm 3 Add Node To Tree

1: **Input:** $\mathcal{T}, \mathbf{x}_{\text{new}}, \mathbf{x}_{\text{closest}}, \mathcal{X}_{\text{near}}$
2: $\mathbf{x}_{\text{min}} = \mathbf{x}_{\text{closest}}$, $c_{\text{min}} = \texttt{cost}(\mathbf{x}_{\text{closest}}) + \texttt{dist}(\mathbf{x}_{\text{closest}}, \mathbf{x}_{\text{new}})$
3: **for** $\mathbf{x}_{\text{near}} \in \mathcal{X}_{\text{near}}$ **do**
4: $\quad c_{\text{new}} = \texttt{cost}(\mathbf{x}_{\text{near}}) + \texttt{dist}(\mathbf{x}_{\text{near}}, \mathbf{x}_{\text{new}})$
5: $\quad$ **if** $c_{\text{new}} < c_{\text{min}}$ and $\texttt{line}(\mathbf{x}_{\text{near}}, \mathbf{x}_{\text{new}}) \in \mathcal{X}_{\text{free}}$ **then**
6: $\quad\quad c_{\text{min}} = c_{\text{new}}$, $\mathbf{x}_{\text{min}} = \mathbf{x}_{\text{near}}$
7: $V_{\mathcal{T}} \leftarrow V_{\mathcal{T}} \cup \{\mathbf{x}_{\text{new}}\}$, $E_{\mathcal{T}} \leftarrow E_{\mathcal{T}} \cup \{\mathbf{x}_{\text{min}}, \mathbf{x}_{\text{new}}\}$

Algorithm 4 Rewire Random Nodes

1: **Input:** $\mathcal{Q}_{\text{r}}, \mathcal{T}$
2: **repeat**
3: $\quad \mathbf{x}_{\text{r}}$=`PopFirst`$(\mathcal{Q}_{\text{r}})$, $\mathcal{X}_{\text{near}}$=`FindNodesNear`$(\mathbf{x}_{\text{r}}, \mathcal{X}_{\text{SI}})$
4: $\quad$ **for** $\mathbf{x}_{\text{near}} \in \mathcal{X}_{\text{near}}$ **do**
5: $\quad\quad c_{\text{old}}$=`cost`$(\mathbf{x}_{\text{near}})$, c_{new}=`cost`$(\mathbf{x}_{\text{r}})$+`dist`$(\mathbf{x}_{\text{r}}, \mathbf{x}_{\text{near}})$
6: $\quad\quad$ **if** $c_{\text{new}} < c_{\text{old}}$ and `line`$(\mathbf{x}_{\text{r}}, \mathbf{x}_{\text{near}}) \in \mathcal{X}_{\text{free}}$ **then**
7: $\quad\quad\quad E_{\mathcal{T}} \leftarrow (E_{\mathcal{T}} \backslash \{\texttt{Parent}(\mathbf{x}_{\text{near}}), \mathbf{x}_{\text{near}}\}) \cup \{\mathbf{x}_{\text{r}}, \mathbf{x}_{\text{near}}\}$
8: $\quad\quad\quad$ `Push` $\mathbf{x}_{\text{near}}$ to the end of $\mathcal{Q}_{\text{r}}$
9: **until** Time is up or $\mathcal{Q}_{\text{r}}$ is empty.

Algorithm 5 Rewire From the Tree Root

1: **Input:** $\mathcal{Q}_{\text{s}}, \mathcal{T}$
2: **if** $\mathcal{Q}_{\text{s}}$ is empty **then**
3: $\quad$ `Push` $\mathbf{x}_0$ to $\mathcal{Q}_{\text{s}}$
4: **repeat**
5: $\quad \mathbf{x}_{\text{s}}$=`PopFirst`$(\mathcal{Q}_{\text{s}})$, $\mathcal{X}_{\text{near}}$=`FindNodesNear`$(\mathbf{x}_{\text{s}}, \mathcal{X}_{\text{SI}})$
6: $\quad$ **for** $\mathbf{x}_{\text{near}} \in \mathcal{X}_{\text{near}}$ **do**
7: $\quad\quad c_{\text{old}}$=`cost`$(\mathbf{x}_{\text{near}})$, c_{new}=`cost`$(\mathbf{x}_{\text{s}})$+`dist`$(\mathbf{x}_{\text{s}}, \mathbf{x}_{\text{near}})$
8: $\quad\quad$ **if** $c_{\text{new}} < c_{\text{old}}$ and `line`$(\mathbf{x}_{\text{s}}, \mathbf{x}_{\text{near}}) \in \mathcal{X}_{\text{free}}$ **then**
9: $\quad\quad\quad E_{\mathcal{T}} \leftarrow (E_{\mathcal{T}} \backslash \{\texttt{Parent}(\mathbf{x}_{\text{near}}), \mathbf{x}_{\text{near}}\}) \cup \{\mathbf{x}_{\text{s}}, \mathbf{x}_{\text{near}}\}$
10: $\quad\quad$ **if** $\mathbf{x}_{\text{near}}$ is not pushed to $\mathcal{Q}_{\text{s}}$ after restarting $\mathcal{Q}_{\text{s}}$ **then**
11: $\quad\quad\quad$ `Push` $\mathbf{x}_{\text{near}}$ to the end of $\mathcal{Q}_{\text{s}}$
12: **until** Time is up or $\mathcal{Q}_{\text{s}}$ is empty.

4.1.2 Rewiring The Tree

Rewiring is done when a node ($\mathbf{x}_{\text{i}}$) gets a lower cost-to-reach c_{i} value by passing from another node instead of its parent. Thus, in our algorithm rewiring should be done around the new node ($\mathbf{x}_{\text{new}}$) that is added (same as RRT*) as well as around the already added nodes (as opposed to RRT*) due to changing the tree root ($\mathbf{x}_0$) and

dynamic obstacles. When $\mathbf{x}_0$ or any dynamic obstacle inside r_o (see Fig 2) changes, we need to rewire large portion of the tree which is done by: 1) Rewiring a random part of the tree (Algorithm 4); 2) Rewiring starting from the tree root (Algorithm 5).

Lines 3-7 and lines 5-9 of Algorithms 4 and 5 rewire neighbor nodes ($\mathcal{X}_{\text{near}}$) of $\mathbf{x}_\text{r}$ and $\mathbf{x}_\text{s}$, respectively. Thus, rewiring is done in $\mathbf{x}_{\text{near}}$ if by changing its current parent to $\mathbf{x}_\text{j}$, c_i for $\mathbf{x}_{\text{near}}$ reduces where $\mathbf{x}_\text{j}$ is $\mathbf{x}_\text{r}$ and $\mathbf{x}_\text{s}$ in Algorithms 4 and 5, respectively. The difference between these algorithms is the focus point of the rewiring. Algorithm 4 rewires random part of the tree starting from nodes around $\mathbf{x}_{\text{rand}}$ or $\mathbf{x}_{\text{closest}}$ that are added to $\mathcal{Q}_\text{r}$ in Algorithm 2. Then if rewiring happens to any $\mathbf{x}_{\text{near}}$, Algorithm 4 adds $\mathbf{x}_{\text{near}}$ to $\mathcal{Q}_\text{r}$ since nodes around $\mathbf{x}_{\text{near}}$ have the potential to get rewired (line 8). However, Algorithm 5 focuses rewiring the tree around $\mathbf{x}_0$ and thus around the agent. Hence, it starts rewiring from $\mathbf{x}_0$ (line 3) and pushes all its neighbor nodes to $\mathcal{Q}_\text{s}$. Then, it continues to push nodes with greater distance from $\mathbf{x}_0$ by popping nodes ($\mathbf{x}_\text{s}$) from $\mathcal{Q}_\text{s}$ (line 5) and pushing $\mathbf{x}_{\text{near}}$ around $\mathbf{x}_\text{s}$ (line 11) when the condition in line 10 is met. Using $\mathcal{Q}_\text{s}$ allows us to rewire the neighbor nodes with the same c_i values from $\mathbf{x}_0$. We refer to the nodes with the same c_is as Rewiring Circle from $\mathbf{x}_0$ (see crossed points in Fig 1). Rewiring continues at each iteration until the condition in lines 9 and 12 of Algorithms 4 and 5 is met, respectively. If time is up and there are still nodes in $\mathcal{Q}_\text{r}$ or $\mathcal{Q}_\text{s}$ that should get rewired, we can continue rewiring in the upcoming iterations.

By pushing nodes with the potential need of rewiring their neighbors to $\mathcal{Q}_\text{r}$, Algorithm 4 intensifies the effect of random sampling in line 2 of Algorithm 2. Also, by the combination of using $\mathcal{Q}_\text{r}$ for random rewiring with focusing sampling inside the Ellipsis that contains the path from $\mathbf{x}_0$ to $\mathbf{x}_{\text{goal}}$, we made it possible to rewire the path to $\mathbf{x}_{\text{goal}}$ very quickly. Note that, when rewiring is done on the path from $\mathbf{x}_0$ to $\mathbf{x}_{\text{goal}}$, we need to update the path again (see Section 4.2). Using $\mathcal{Q}_\text{s}$ in Algorithm 5, allows us to grow the Rewiring Circle between iterations. As the Rewiring Circle grows (supplemental video 00:40) every node inside this circle is rewired. Thus, on the already expanded tree, a minimum path to every node in the circle is created due to the circle that grows from $\mathbf{x}_0$. Note that, Algorithm 5 only rewires on the already expanded tree and since the tree is not complete in the beginning, it is possible that when a new node is added to the tree, Algorithm 4 creates paths with lower c_is to the nodes in the circle. $\mathcal{Q}_s$ gets restarted when $\mathbf{x}_0$ is changed or a dynamic obstacle blocks a node with lower c_i than c_is of the nodes on the Rewiring Circle. In Algorithm 5, restarting $\mathcal{Q}_\text{s}$ means we need to rewire nodes from $\mathbf{x}_0$ again (line 10).

4.2 Path Planning

Line 7 in Algorithm 1 uses Algorithm 6 to plan a k-step path from $\mathbf{x}_0$. Algorithm 6 plans the path in two ways: 1) when tree reaches $\mathbf{x}_{\text{goal}}$ (lines 2-4) which happens when we expand the tree using Algorithm 3; 2) when the tree has not reached $\mathbf{x}_{\text{goal}}$ (lines 6-10). In the first way, the path from $\mathbf{x}_{\text{goal}}$ up to $\mathbf{x}_0$ is in the tree and thus we only need to update the path (line 3) when rewiring the path is done using rewiring algorithms in Section 4.1.2. In the second way, we plan a path to get as close as possible to $\mathbf{x}_{\text{goal}}$ on the tree. We use cost function $f_\text{i} = c_\text{i} + h_\text{i}$ to get close to $\mathbf{x}_{\text{goal}}$ as well as to take a short path on the growing tree. Thus, using f_i can trap us in local minima. To prevent trapping in local minima and enable path planning to visit other branches, we plan a k-step path using f_i at each iteration (lines 6,7) and block the already seen nodes (line 10) by setting their h_is to infinity. When path planning reaches a node that could not go any further (line 8), we return the planned path and block that node (lines 9,10). Then, we update the best already found path if the planned path leads us to a location closer to $\mathbf{x}_{\text{goal}}$ (line 11). However, the agent follows the best path if it leads to some place closer than the current place of the agent (line 12).

Algorithm 6 Plan a Path for k Steps

1: **Input:** $\mathcal{T}$, $\mathbf{x}_{\text{goal}}$
2: **if** Tree has reached $\mathbf{x}_{\text{goal}}$ **then**
3: Update `path` from $\mathbf{x}_{\text{goal}}$ to $\mathbf{x}_0$ if the `path` is rewired
4: $(\mathbf{x}_0, ..., \mathbf{x}_\text{k}) \leftarrow (\mathbf{x}_0, ..., \mathbf{x}_{\text{goal}})$
5: **else**
6: **for** $\mathbf{x}_\text{i} \in (\mathbf{x}_1, \mathbf{x}_2, ..., \mathbf{x}_\text{k})$ **do**
7: $\mathbf{x}_\text{i}$=child of $\mathbf{x}_{\text{i-1}}$ with minimum f_c=`cost`$(\mathbf{x}_\text{c})$+`H`$(\mathbf{x}_\text{c})$
8: **if** $\mathbf{x}_\text{i}$ is leaf or its children are blocked **then**
9: $(\mathbf{x}_0^{'}, ..., \mathbf{x}_\text{k}^{'}) \leftarrow (\mathbf{x}_0, ..., \mathbf{x}_\text{i})$
10: Block $\mathbf{x}_\text{i}$ and Break;
11: Update best `path` with $(\mathbf{x}_0^{'}, ..., \mathbf{x}_\text{k}^{'})$ if necessary
12: $(\mathbf{x}_0, ..., \mathbf{x}_\text{k}) \leftarrow$ choose to stay in $\mathbf{x}_0$ or follow best `path`
13: **return** $(\mathbf{x}_0, ..., \mathbf{x}_\text{k})$

Note that, `H`$(\mathbf{x}_\text{i})$ returns infinity if $\mathbf{x}_\text{i}$ is blocked and $\mathbf{x}_{\text{goal}}$ has not been changed, i.e. $\mathbf{x}_\text{i}$ is already visited for the current $\mathbf{x}_{\text{goal}}$. Otherwise, if $\mathbf{x}_{\text{goal}}$ has been changed or $\mathbf{x}_\text{i}$ has not been visited for the current $\mathbf{x}_{\text{goal}}$, `H`$(\mathbf{x}_\text{i}) = \|\mathbf{x}_\text{i} - \mathbf{x}_{\text{goal}}\|_2$. Every node has a chance to get visited again, i.e. all its ancestors get unblocked, when another unblocked node is added as its child by rewiring or adding a node. It should be noted that when the tree has reached $\mathbf{x}_{\text{goal}}$ and the path gets blocked by an obstacle, we follow the path up to the obstacle until rewiring finds another path or the path gets cleared.

5 Simulations

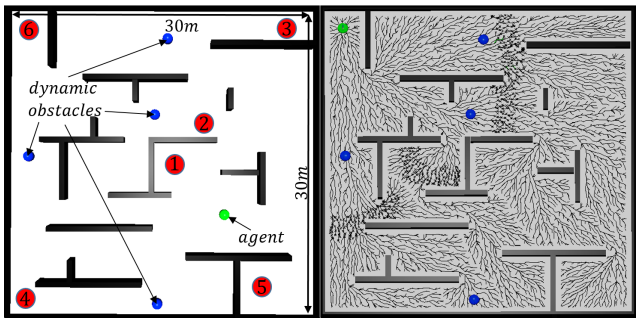


Figure 3: *Simulation environment (left) and the tree expanded by RT-RRT* to cover this environment (right). The numbered circles denote different goal areas in the environment.*

In this section we compare our algorithm, RT-RRT*, with CL-RRT [Luders et al. 2010] in two different scenarios. First, we find out the number of iterations (planning step and taking an action) needed to find a path to the various goals in Fig 3. Secondly, we find out the length of the paths taken between the consecutive goal points of Fig 3. The dynamic obstacles are motionless in the simulation to make a fair comparison with CL-RRT. However, our supplementary video shows how RT-RRT* reacts to the movements of the dynamic obstacles (supplemental video 00:15). The agent and dynamic obstacles have the relatively small radii of $0.5m$. The tree expanded by RT-RRT* is demonstrated in Fig 3. This tree covers the whole environment with approximately 7000 nodes. We had a grid of size $2m \times 2m$ which divides the environment into 225 squares. The k_{max} and r_s in Algorithm 2 are set to 5 and 0.5 m respectively. α and β in (1) are 0.1 and 2, correspondingly. Also, we set r_o, r_b and r_g of Fig 2 to $10m$, $1.5m$ and $0.5m$, respectively. The limited time for Tree Expansion-and-Rewiring is set to 10 ms and we set k in Algorithm 1 to 100. In order to have a fair comparison with

CL-RRT, we had a disturbance free model and used C-PBP as the closed loop controller of the CL-RRT algorithm as well. This way the differences in the sampling schemes become evident. In CL-RRT, The number of the nodes is limited to 1000 for real-time use. Also, We plan 10 steps ahead with CL-RRT and use 5 as the nearest neighbor parameter.

We ran each algorithm for each scenario 10 times and averaged the results. The average number of iterations required to find the path to each goal was 8.26 for our method and 87.54 for CL-RRT. The average number of the iterations needed to find the path to $G1$ was approximately the same for both of the algorithms since both of them start growing the tree from a scratch. However, as opposed to CL-RRT, our method needs fewer iterations as the tree grows in the environment. Particularly, in RT-RRT* when the tree is grown to the whole area, the path is fast to find, e.g. the average number of the iterations for finding paths to $G4$, $G5$ and $G6$ are 1.00, 4.27 and 1.00 for RT-RRT*, respectively, whereas for CL-RRT they are of the order of 100. This happens because nothing is pruned from the tree unlike in CL-RRT (supplemental video 1:30). In addition, the average length of a path taken to the different goals was 27.12 for our method and 72.67 for CL-RRT. We can see that RT-RRT* needs less iterations and produces paths with smaller length. The smaller length of the path is caused by the rewiring operation which is present only in RRT* based algorithms. We also observed that using 500 iterations CL-RRT failed to find the path to $G5$ with 65 % chance and to $G3$ with 10 % chance because of the narrow passages, whereas RT-RRT* never fails to find a path as the tree is expanding.

6 Conclusions

In this paper we introduced the first real-time version of RRT* and informed RRT*. The real-time capability was achieved by interleaving the path planning with the tree expansion and rewiring. Furthermore, we move the tree root with the agent to retain the tree instead of building it anew at every iteration. The tree continues to grow until it covers the environment. We also introduced two modes of rewiring for having shorter paths in the large tree with a limited number of the nodes. These are: 1) rewiring starting from the root, and 2) rewiring random parts of the tree. The first one creates a growing circle centered at the agent. In this process every node inside the circle is rewired, and this circle most frequently rewires nodes around the agent and thus the tree root. The second one is done using both focused and uniform sampling, but in patches instead of just one node. We tested our algorithm against CL-RRT which can be considered the state-of-the-art in RRT-based real-time path planning. Our simulations show that the combination of retaining the tree with the two methods of rewiring the large tree in RT-RRT* enables the algorithm to find shorter paths with less iterations to one or multiple goal points. One should note that, for the sake of simplicity and memory, we used a grid to speed up our neighboring node search. Some more sophisticated spatial index such as a KD-tree would speed up the search even more and allow using bigger sampling budgets.

RT-RRT* has its limitations, e.g. it requires a large memory capacity because the whole tree is stored at all times. One of the major limitations of our algorithm is that it only works in a bounded environment. The focused sampling inside an ellipsoid works somewhat in an unrestricted environment but the rewiring suffers if the distances are large. Thus, the challenges of unbounded and large distance environments remain to be addressed.

We used C-PBP to move the agent on the planned path and separate the path planning from the motion planning. Although C-PBP enables the agent to follow the planned path smoothly and plan motions around the obstacles, it regrettably does not provide us with a model of the obstacles. That is why we made assumptions about which objects are obstacles and resorted to circumventing the obstacles with a given minimum distance. We deem extracting an obstacle model a fruitful research direction for the future.

Acknowledgements

This work has been supported by TEKES (The Finnish Funding Agency For Innovation).

References

BRUCE, J., AND VELOSO, M. 2002. Real-time randomized path planning for robot navigation. In *IROS*, IEEE.

CANNON, J., ROSE, K., AND RUML, W. 2012. Real-Time Motion Planning with Dynamic Obstacles. In *Symposium on Combinatorial Search*.

GAMMELL, D, J., SRINIVASA, S, S., BARFOOT, AND D, T. 2014. Informed RRT*: Optimal Sampling-based Path Planning Focused via Direct Sampling of an Admissible Ellipsoidal Heuristic. In *IROS*, IEEE.

GUTMANN, J.-S., FUKUCHI, M., AND FUJITA, M. 2005. Real-time path planning for humanoid robot navigation. In *IJCAI*.

HÄMÄLÄINEN, P., RAJAMÄKI, J., AND LIU, C. K. 2015. Online Control of Simulated Humanoids Using Particle Belief Propagation. In *Proc. SIGGRAPH '15*, ACM.

KARAMAN, S., AND FRAZZOLI, E. 2011. Sampling-based Algorithms for Optimal Motion Planning. *Int. J. Rob. Res.*.

KAVRAKI, L. E., ŠVESTKA, P., LATOMBE, J.-C., AND OVERMARS, M. H. 1996. Probabilistic roadmaps for path planning in high-dimensional configuration spaces. *IEEE Trans. Robot. Autom.*.

KHATIB, O. 1986. Real-time obstacle avoidance for manipulators and mobile robots. *The International Journal of Robotics Research*.

LAVALLE, S. M. 1998. Rapidly-Exploring Random Trees: A new Tool for Path Planning.

LUDERS, B. D., KARAMAN, S., FRAZZOLI, E., AND HOW, J. P. 2010. Bounds on tracking error using closed-loop rapidly-exploring random trees. In *American Control Conference*, IEEE.

OTTE, M., AND FRAZZOLI, E. 2015. RRT$^{\mathrm{X}}$: Real-time motion planning/replanning for environments with unpredictable obstacles. In *Algorithmic Foundations of Robotics XI*. Springer.

RASTGOO, M. N., NAKISA, B., NASRUDIN, M. F., NAZRI, A., AND ZAKREE, M. 2014. A critical evaluation of literature on robot path planning in dynamic environment. *Journal of Theoretical & Applied Information Technology*.

SONG, S., LIU, W., WEI, R., XING, W., AND REN, C. 2014. Path planning directed motion control of virtual humans in complex environments. *Journal of Visual Languages & Computing*.

STURTEVANT, N. R., BULITKO, V., AND BJÖRNSSON, Y. 2010. On learning in agent-centered search. In *AAMAS*.

SUD, A., ANDERSEN, E., CURTIS, S., LIN, M., AND MANOCHA, D. 2008. Real-time path planning for virtual agents in dynamic environments. In *ACM SIGGRAPH 2008 Classes*, ACM.