Path Finding in Unknown Environment Based on Exploration Agents

Xugang Ye

Applied Mathematics and Statistics, Johns Hopkins University
Baltimore, MD, 21218, USA

Abstract
Path finding is an interesting and challenging topic in the field of Artificial Intelligence. It has received extensive research attention during the last three decades. Most of the early work was focused on the geometric path finding problems in which the shortest paths connecting a set of origins and a set of goals in a geometric space populated by a finite number of previously known static polygonal obstacles are supposed to be found. Relatively recently, some researchers addressed the problem of path finding on the planetary surface. A feature of this problem is that the globally reliable prior terrain information is not available, or only partially available. Hence, the navigation decisions are highly dependent on the accurate local information dynamically and incrementally collected by the rovers themselves. Since the 1990’s several replanning algorithms have been developed for navigating a single ground mobile robot in a changing, or partially known, or unknown environment. In order to improve the efficiency and reliability, the use of multiple agents are suggested in very recent literature. Motivated by this trend, this paper aims to find efficient and effective algorithm for commanding a team of ground mobile robots to start from a single base, move in an unknown environment, and finally find the target location in a swift manner. Meanwhile, based on the acquired environmental knowledge, a path from the base to the target location is constructed such that the traveling cost (or defined as length in general sense) of this path is as small as possible. In this paper, we first study a generic single-agent exploration mechanism and its practical implementation; we then extend the mechanism into the coordination methods for multiple agents. Both the theoretical and the numerical results are presented.

Key words: Path Finding, Unknown Environment, Multiple Agents, and Optimization.

1. An Overview of Literature
1.1. Classical Topic
In general, path finding is concerned with finding paths connecting different locations in an environment. If the environment takes the form of a network or a graph, in which the nodes or vertices are defined as the locations and the weights of the arcs or edges are defined as the transition costs, then path finding falls into the range of the classical shortest path problems (See Cormen et al., 2001 [1] and Ahuja et al., [2] for a comprehensive survey). In the field of Artificial Intelligence (AI), path finding deals with computing desired paths in a geometric space embedded with forbidden regions or risk areas. One of the most fundamental geometric path finding problems is to find a shortest path in a plane populated by a finite number of pre-known static polygonal obstacles, without passing any
interior point of any obstacle. This problem was initially solved by constructing a visibility graph and invoking a shortest path algorithm on the graph (see Lozano-Perez and Wesley, 1979 [3]). This approach fueled intensive research on computing visibility graphs. For the time complexity of computing a visibility graph see Welzl, 1985 [4] for a $O(n^2)$ time algorithm, where $n$ is the total number of obstacle vertices and see Ghosh and Mount, 1991 [5] for a $O(n\log n + e)$ time algorithm, where $n$ is the total number of obstacle vertices, $e$ is the number of edges in the graph. Note that the visibility graph can have $O(n^2)$ edges in the worst case; the currently known heap implementations of the Dijkstra algorithm that solves the single source shortest path problem for a directed graph with nonnegative edge weights contains $O(e)$ term in time complexity (e.g., see [1] for binary heap with time complexity $O((e + n)\log n)$; see Fredman, 1987 [6] and [1] for Fibonacci heap with time complexity $O(e + n\log n)$; and see Ahuja, 1990 [7] for a combination of Fibonacci heap and radix heap with the time complexity $O(e + n\sqrt{\log C})$, where $C$ is the upper bound of the nonnegative integer edge weights), hence the time complexity of this approach is $O(n^2)$ in the worst case. An improvement can be found in Hershberger and Suri, 1993 [8], in which a method that combines the shortest path map approach (see Mitchell et al., 1993 [9]) and a quad-tree-style subdivision of the plane (see Bern et al., 1990 [10]) is proposed, with a time complexity $O(n\log^2 n)$ and a space complexity $O(n\log n)$. When the problem is extended into three dimensional, one can find in Canny and Reif, 1987 [11] that the extended problem is NP-hard. For a survey on solving geometric shortest path problems in 3-D settings see Mitchell, 1998 [12]. Efforts have been made to compute the near-shortest paths for the same geometric path finding problem in 2-D setting, with the time complexities smaller than that required to compute a shortest path. The concept of $\varepsilon$-shortest path was proposed to describe a feasible path with the length at most $(1+\varepsilon)$ times the length of the shortest path, where $\varepsilon > 0$. Some representative methods include the retraction algorithm (see O’Dunlaing and Yap, 1985 [13]) based on the Voronoi diagram (see Aurenhammer, 1991 [14]) of the terrain, the trapezoidal decomposition of free space via plane sweep technique (see Preparata and Shmos, 1985 [15]), the hierarchical triangulation of the simple terrain (see Kirkpatrick, 1983 [16] and Zhu and Latombe, 1991 [17]), the bug-like local search algorithms (see Lumelsky and Stepanov, 1987 [18]), and the artificial potential field methods (see Koditschek. 1989 [19]) etc.

1.2. Recent Developments
1.2.1. Trend
In late 1990’s, path finding found its practical applications in computer games and the navigation systems for autonomous robots. To practitioners, the challenges of incorporating the existing geometric shortest path algorithms into the real-world application systems are overwhelmingly strong. For example, other than just considering the static polygonal obstacles on a perfectly “flat” land, the practical path finding algorithms must be able to deal with very complicated obstacle shapes, non-flat landscapes, and, if possible, the dynamic environment. The cost of a path may also depend on more “general” factors than the Euclidian distance. Those factors may include the type of areas passed through, slopes, turning angles etc (see Chen, 1996 [20]). Many known geometric shortest path algorithms are very environment-specific and depend on sophisticated data structures and geometric
procedures (see Latombe, 1991 [21] and Hwang and Ahuja, 1992 [22] for good surveys). Hence, it is necessary to develop simple-to-implement yet reasonably efficient algorithms that work for more “general” path finding systems. It is also highly desirable to develop such algorithms that are compatible with some “standard” input (e.g., terrain matrices). Under this requirement, a practically useful path finding method should include at least two features: 1) flexible terrain modeling, 2) efficient and effective search mechanism.

1.2.2. Terrain Modeling

Terrain modeling is the preprocessing phase of the path finding. For the simple case like polygonal forbidden regions in a flat plane, the exact representation of the world could be used. Any method of this type needs to have a special data structure to store the obstacle information (e.g., vertices and edges, or facets if the space is three dimensional). In classical literature, methods that establish exact representation of the world include visibility graph, Voronoi diagram, and triangulation etc. Despite the accuracy, this type of methods has little practical application. The more flexible mapping technique is the space discretization, which extensively appears in the development of real-time strategy computer games and the recent path finding literature. The central idea is to decompose the world into mutually exclusive cells regardless of the obstacles and the area types. For each cell, the reachability or the traversability is defined deterministically or even probabilistically. The transition cost from one cell to an adjacent cell is also defined regarding the effort of the agent. For the agent, if a state is defined as the cell in which the agent is located, then a path from the origin to the goal can be defined as a sequence of non-separable feasible state transitions from the initial state to the target state. Hence, the world is translated into a “labeled” grid network. The most notable advantage is that the obstacles or forbidden regions can be coded as the union of specifically labeled cells without considering their intrinsic geometric structures. The higher the grid resolution, the higher the map accuracy. Another advantage is that a grid network can be easily extended if the size of the map needs to be increased. This is usually done with an associated coordinate system. Typically, a cell can be a square, or a hexagon, or a triangle. One can find in literature that the eight-connected square grid network is very popular due to its easy implementation and relatively efficient memory requirement. For more details on the concepts and algorithms for building a grid network and its coordinate system, see Grunbaum and Shephard, 1986 [23] and Chavey, 1989 [24]. An excellent web source can be found at Patel, 2006 [25]. Under some circumstance, some more complicated non-uniform grid networks can be used. For example, when a flat plane is very sparsely embedded with some graphically complicated obstacles, a quadtree (see Samet, 1988 [26]) is more efficient than a regular square grid network (and also a visibility graph, or a Voronoi diagram). The reason is that the large empty areas are only coded with very low resolution, hence, both the storage and the searching scale is reduced. The price is the considerably increased complexity in data structure (e.g., see Kambhampati and Davis, 1986 [27]). In effect, a disadvantage of quadtree is that a path found with such map representation is usually jagged. An improvement is to use a new representation called “framed” quadtree (e.g., see Chen et al., 1997 [28] and Yahja et al., 1998 [29]), which is a modified version of quadtree. In framed quadtree, cells of the highest resolution are added around the perimeter of each quadtree region. The non-separable intermediate state transition is redefined as the shift from one cell of the highest resolution to a
neighbor cell of the highest resolution. It has been shown [29] that the path quality can be significantly improved if the quadtree is replaced by its framed version. However, since the grid network generated by a framed quadtree could be much more dense (i.e. a node has more incident arcs) than that generated by the corresponding quadtree, a path finding search algorithm executed on such network could have much higher time complexity. Moreover, when the quadtree or framed quadtree is extended to represent an unknown world, a dynamic subdivision procedure is involved. If the environment is uniformly, highly cluttered (e.g., a fractal terrain), the framed quadtree can require much more memory than the regular square grid network [29]. To the author’s knowledge, there is no extended work of using quadtree or framed quadtree for multiple agents due to the inherent high complexity.

1.2.3. Search Algorithms

Once the terrain is properly modeled, the second phase of path finding is to invoke an efficient search algorithm. In the network flow community, path finding algorithms are typically classified into two groups: label setting (e.g., the Dijikstra algorithm) and label correcting (e.g., the Bellman-Ford algorithm and the Floyd-Washall algorithm). The second group is more general in the sense that no label is viewed as permanent until the termination. Although the path finding algorithms in this community were initially designed for finding shortest path/paths in a finite graph/network, the generic search mechanism makes them suitable, after modifications, for target searching in an infinite grid world in which each node only has a few neighbors geographically close to it. This prompts the counterparts in the AI community. In this community, A* algorithm (see Hart et al., 1968 [30], Nilsson, 1980 [31], Pearl, 1984 [32], Russell, 2003 [33], Lester, 2005 [34], and Patel, 2006 [35]) plays one of the most fundamental roles in the intelligent search strategies. Generically, with proper implementation, the A* algorithm can be applied to omniscient world, partially known world, unknown world, and dynamic world. Provably, the A* algorithm is much more efficient than the Dijikstra algorithm for finding a least-cost path from an origin to a goal in a network that is embedded in a Euclidean space. The reason is that the branching factor of the search tree is reduced by the “informed” heuristic function, which estimates the remaining “distance” to the goal and hence the search progress. Practically, the A* algorithm can be extended into diverse variants upon the different needs. The time complexity of the A* algorithm depends on the heuristic. In the worst case, the number of nodes expanded (i.e. the exploration scale) is exponential in the “length” of the solution (the shortest path), but it is polynomial when the error of the heuristic is bounded by a logarithm of the “perfect” heuristic [33]. When applied to a dense network, the memory requirement of the A* algorithm could be extremely high. Several variants of the A* algorithm designed for this problem include iterative deepening A* (IDA*) (see Korf, 1985 [36] and Nilsson, 1995 [37]), memory-bounded A* (MA*) [33, 37] and simplified memory bounded A* [33, 37]. When the environment of path finding is dynamic, the A* algorithm is implemented as its replanning version (or dynamic version). Suppose the A* algorithm finds a path for an agent to travel along given an initial map. When the agent moves to somewhere in the middle of the original path, the map is updated. This may affect the legality of the original path, hence, a replanning is necessary. The earliest replanning algorithm is brute-force like (see Zelinsky, 1992 [38]). A significant improvement is the Focused D*
Algorithm (see Stentz, 1995 [39]), which is the “informed” version of the D* Algorithm (or called dynamic A* algorithm) (see Stentz, 1994 [40]). D* algorithm is a replanning version of the A* algorithm yet far more efficient than the brute-force counterpart [40]. A functionally same but algorithmically different algorithm is the D* Lite algorithm (See Koenig and Likhachev, 2002 [41]), which is the “reversed” version of the LPA* algorithm (See Koenig and Likhachev, 2002 [42]). The LPA* algorithm is also an “informed” version of an earlier algorithm called DynamicSWSF-FP (see Ramalingam and Reps, 1996 [43]). Replanning algorithms were designed for navigating a single traveling agent. They are good for a known map with dynamic perturbations. They are also good for an environment with a small unknown potion of area. For an entire unknown environment, the extensive replanning work may cancel their advantage over the A* algorithm. Moreover, in an infinitely large grid network, it’s hard to decide a range for initialization. To the author’s knowledge, there is no report on replanning algorithms for multiple agents.

1.2.4. Multi-agent System

The main reason to consider a multi-agent system other than a single-agent system in many applications is that a multi-agent system can be more fault-tolerant (through redundancy) and efficient (through parallelism). If a given global task is in nature functionally and sequentially dividable, then it is possible for a team of agents to cooperate to accomplish this task. The difficulty arises in coordinating those agents. The methods of coordinating multiple agents to accomplish a single, global task can be roughly classified as the centralized optimization and the distributed (i.e. decentralized) optimization. A centralized model usually takes the form of mathematical programming. The principal advantage is that an optimal plan can be produced if the relevant information for decision-making is enough and accurate. However, such model is brittle due to the possibility of the environmental change, agent failure, and task alteration etc. Since a centralized model is usually computationally expensive to solve, the replanning could be slow, hence, a multi-agent system controlled by such mechanism is sluggish in responses to the unexpected perturbations or disruptions. As an alternative, a distributed model is more realistic. In a distributed model, each agent is viewed as a decision-maker, which is capable of collecting information and performing computations relevant to its behaviors. Hence, it is possible for each agent to act not only concurrently but also quickly. Compared with the centralized coordination, each agent has more freedom. However, conflicts always exist. Hence communication among the agents is necessary. The advantage of the distributed optimization is that the communication among the agents can be treated locally (i.e. the behaviors of an agent is affected by those only in its “vicinity”), hence, a perturbation may only propagate very locally. As same with the centralized coordination mechanism, in order to have an overview of the global progress, a central database is needed for monitoring the system performance. This database has an interface with each agent such that the efforts made by each agent can be referred by others. The central database can help to reduce the repeated efforts, and thus, the chance of repeated wrong decisions. Although the distributed approaches in general only return a suboptimal system performance, they are well accepted by applied mathematicians, operations research practitioners, and computer scientists as the experimental yet practical ways for handling complicated problems. The past decade has witnessed a growing focus on multi-agent systems. Two intuitive summaries are Mataric, 1995 [44] and Dias and
Stentz, 2000 [45]. In very recent literature, some auction-based algorithms have been designed for the multi-robot routing problem. The central idea is to find the solution to the embedded task allocation problem through multiple rounds of bidding (and if possible, subcontracting). If the environment is known, then the routing problem is similar to the classical multi-vehicle routing problem (see Fisher, 1995 [46]); if the environment is unknown the problem has its dynamic version (see Psaraftis, 1995 [47]). The success in numerical experiments prompts the further investigations into the nature of the market mechanism. Some representative work include Gerkey and Mataric, 2002 [48], Zlot et al., 2002 [49], Goldberg et al., 2003 [50], Tovey et al., 2005 [51], Lagoudakis et al., [52], and Koenig, et al., 2006 [53] etc.

2. Problem and Formulation

2.1. Problem Description

Based on the study of the literature, this paper aims to look at the path finding from the multi-agent point of view. The problem under investigation can be roughly stated as commanding a team of ground mobile robots to start from a single base, move in an unknown environment, and finally find the target location in a swift manner. Meanwhile, based on the acquired environmental knowledge, a path from the base to the target location is constructed such that the traveling cost (or length) of this path is as small as possible. Imagine on a planetary surface multiple identical agile rovers equipped with obstacle detection sensors are sent from a base to explore the world and find a path to a target location (Figure 2.1.1). Each rover collects some terrain information as it moves and evaluates the reachability or traversability of its vicinity. The information is transmitted to the base for a central processing unit to synthesize and analyze. Once a rover reaches the target location, the central processing unit immediately figures out a relatively short path from the base to the target location and sends a special ground mobile robot to the target location for some scientific missions (e.g. drilling, collecting samples, and deploying equipments etc). The scientific robot travels along the planned path, reaches the goal, finishes the missions, and returns via the same path yet in reverse direction. Based on the cumulative terrain knowledge, the central processing unit also plans a short path for each rover to return. The main reason for proposing the use of multiple explorers is that a map thus constructed is much more reliable than that computed from the high altitude images according to the current technology.

2.2. Formulation

We are now ready to formulate the problem. Suppose that the world is represented by an eight-connected grid network as shown in the Figure 2.2.1. The grid size, say $\delta \in \mathbb{R}^1$, $\delta > 0$, is defined according to the detection range, say $\rho \in \mathbb{R}^1$, $\rho > 0$, of each exploration agent (assuming that each one has the same detection range) as $\delta = \frac{\sqrt{2}}{3} \rho$. The center of a cell is called a node, which is denoted as $n_{ij}$, $i \in \mathbb{Z}^1$, $j \in \mathbb{Z}^2$, it has eight neighbor nodes denoted as $\text{Neighbor}(n_{ij}) = \{ n_{i+1,j-1}, n_{i-1,j}, n_{i+1,j+1}, n_{i,j-1}, n_{i,j+1}, n_{i-1,j-1}, n_{i+1,j}, n_{i+1,j+1} \}$. The traveling cost from a node $u$ to a neighbor $u'$ is denoted as $C(u, u') \in \mathbb{R}^1$, $C(u, u') > 0$. In the simplest case, it is a linear function of the Euclidean distance, say $d(u, u') \in \mathbb{R}^1$, $d(u,$
Figure 2.2.1. Grid representation of the terrain (eight-connected)

Figure 2.1.1. Multi-agent exploration
from a node in the form of ),( \( u' \) > 0, from \( u \) to \( u' \). We can write \( C(u, u') = a(u') \cdot d(u, u') \), where \( a(u') \in \mathbb{R}^1 \), \( a(u') \geq 1 \) denotes the reachability of the node \( u' \). \( a(u') = 1 \) represents the ideal travel, \( a(u') > 1 \) represents the travel with difficulty, and a threshold, say \( a_o \in \mathbb{R}^1 \), \( a_o > 1 \), can be defined such that when \( a(u') \geq a_o \), the node \( u' \) is unreachable. A one-step transition is a travel from a node to one of its reachable neighbors. A path from a node \( u_1 \) to another node \( u_k \) consists of a finite sequence of one-step transitions, it can be denoted as \( P_{u_i, u_k} = u_1 \sim u_2 \sim \cdots \sim u_{k-1} \sim u_k \), where \( k \in \mathbb{Z}^1, k \geq 2 \); \( u_i \neq u_j \) if \( i \neq j \) for any \( i, j \in \{1, 2, \ldots, k\} \), whereby the traveling cost of the path can be defined as \( C(P_{u_i, u_k}) = \sum_{i=1}^{k-1} a(u_i, u_{i+1}) \cdot d(u_i, u_{i+1}) \). A base (or depot) is the set of all the nodes that are within a circle of center \( s \) and radius \( R \), where \( R \in \mathbb{R}^1, R > r \), that is \( B(s, R) = \{n_{ij} \mid d(s, n_{ij}) < R\} \), where \( d(s, n_{ij}) \) is the Euclidean distance from \( s \) to \( n_{ij} \). Suppose the target is denoted by the node \( t \).

Note that the environment is unknown outside the base, there is no global data on the grid network, hence, the classical linear programming formulations (e.g., see [2] and Xie and Xing, 2000 [54] for min-cost flow models) of the shortest path problems for finite network are not appropriate. Since the exploration decision of each agent is made based on its cumulative knowledge of the world, it’s natural to describe the exploration of a single agent in a dynamic manner (as the model (2.2.1-2.2.8) below says).

\[
\text{Min} \left[ \sum_{k=0}^{N-1} C(P_{v_k, v_{k+1}}), C(P_{v_{N-1}, t}) \right] \tag{2.2.1}
\]

Subject to

\[
P_{v_0, t} = v_0 \sim a_0 \sim \cdots \sim a_r \sim t, a_i \in E_N, i = 0, \ldots, r; \tag{2.2.2}
\]

\[
E_{k+1} = E_k \cup \{v_{k+1}\}, k = 0, 1, \ldots, N-1; \tag{2.2.3}
\]

\[
P_{v_k, v_{k+1}} = v_k \sim u_0 \sim \cdots \sim u_{r_k} \sim v_{k+1}, u_i \in E_k, i = 0, \ldots, r_k, k = 0, 1, \ldots, N-1; \tag{2.2.4}
\]

\[
v_{k+1} \in \{u' \mid u' \in \text{Neighbor}(u), u \in E_k, u' \notin E_k, a(u') < a_o\}, k = 0, 1, \ldots, N-1; \tag{2.2.5}
\]

\[
v_N = t; \tag{2.2.6}
\]

\[
v_0 \in B_0; \tag{2.2.7}
\]

\[
E_0 \subseteq B(s, R). \tag{2.2.8}
\]

The key decision variables are \( v_k \)'s, \( k = 1, 2, \ldots, N-1 \). The objective functions in (2.2.1) are to minimize the total traveling cost of the agent and the cost of the path from the base to the target retrieved from the final explored region (as the constraint (2.2.2) indicates). Each term in the first objective function, in the form of \( C(P_{v_k, v_{k+1}}) \), is the cost of some path \( P_{v_k, v_{k+1}} \) the agent takes from \( v_k \) to \( v_{k+1} \). Note from the constraint group (2.2.4) that the path \( P_{v_k, v_{k+1}} \) is picked up from the explored region up to the \( k \)th
stage. The number of the stages for planning, say $N$, is a variable depending on when the target is reached. The constraint (2.2.6) says the final state of the agent must be the destination node $t$. The constraints (2.2.7-2.2.8) say that the agent starts from the base. The constraint groups (2.2.3-2.2.5) say that the exploration is based on the cumulative knowledge of the world and by the constraint group (2.2.3) the knowledge is strictly monotonically increasing through the local accumulation.

If there are $m > 1$ agents, a synchronized multi-agent exploration model can be constructed by extending the model (2.2.1-2.2.8). The objectives include minimizing the effort of the agent that finds the target first and minimizing the length of the base-target path retrieved from the final map. Hence, the team finds a good path to the target in a swift manner. The model is listed as follows (2.2.9-2.2.16) with the key decision variables $v^{(j)}_k$, $k = 1, 2, \cdots, N-1$.

$$\text{Min } \left[ \sum_{k=0}^{N-1} C(P^{(j)}_{v^k_0, v^k_1}), C(P_{s_i}) \right]$$

Subject to

$$P_{s_i} = s \sim \alpha_0 \sim \cdots \sim \alpha_t \in E_N, i = 0, \cdots, r; \quad (2.2.10)$$

$$E_{k+1} = E_k \cup \{v^{(j)}_{k+1}, j = 1, 2, \cdots, m\}, k = 0, 1, \cdots, N-1, j = 1, 2, \cdots, m; \quad (2.2.11)$$

$$P_{v^{(j)}_{k+1}, v^{(j)}_{k+1}} = v^{(j)}_k \sim u^{(j)}_0 \sim \cdots \sim u^{(j)}_{v^{(j)}_{k+1}} \sim v^{(j)}_{k+1}, \quad u^{(j)}_i \in E_k, i = 0, \cdots, r^{(j)}_k, k = 0, 1, \cdots, N-1, \quad (2.2.12)$$

$$j = 1, 2, \cdots, m;$$

$$v^{(j)}_{k+1} \in \{u' \mid u' \in \text{Neighbor}(u), u \in E_k, u' \notin E_k, a(u') < a_x\}, k = 0, 1, \cdots, N-1, \quad (2.2.13)$$

$$j = 1, 2, \cdots, m;$$

$$v^{(j)}_{N+1} = t, l \in \{1, 2, \cdots, m\}; \quad (2.2.14)$$

$$v^{(j)}_0 \in E_0, j = 1, 2, \cdots, m; \quad (2.2.15)$$

$$E_0 \subseteq B(s, R). \quad (2.2.16)$$

Both the model (2.2.1-2.2.8) and the model (2.2.9-2.2.16) can be viewed as the mathematical formulation of some multi-stage decision planning process. According to the single-agent model, the agent selects a subtarget in each planning stage. In each stage, based on the current knowledge of the world, a path from the agent’s current location to the subtarget is determined. The agent moves along the path, reaches the subtarget, and then begins its next stage of decision planning. Meanwhile, when the agent reaches the subtarget, the environmental information of its vicinity is collected and added to the current knowledge. Hence, the map is incrementally updated. In the synchronized multi-agent model, the multi-stage decision planning for a single agent is extended into that for a team. In each stage, a subtarget is planned for each agent if possible. Based on the current knowledge of the world,
for each agent, a path from its current location to its assigned subtarget is determined. When all the agents reach their respective subtargets, the next stage of decision planning begins. In the mean time, based on the collected the environmental information on each agent’s vicinity, the current map is incrementally updated. If each agent is capable of computing, then distributed computation architecture can be established such that the computation for determining the path from its current location to its assigned subtarget and the associated traveling cost can be performed concurrently.

Note that the model (2.2.9-2.2.16) is a fault tolerant formulation since when some agent malfunctions, the only indication is that the contribution of this agent to the accumulation of the system’s knowledge of the world stops, hence, the remaining agents can still move on in the same manner of the multi-stage decision planning. The model also supports the parallelism since each agent moves largely independently. If each agent is viewed as a decision maker, then entire freedom for each agent usually likely leads to conflicts throughout the team. Therefore, centralized control and coordination, in some degree, is needed. In the later sections, we will find that to resolve the conflicts that stem from the distributed optimization, some market mechanism plays a key role, it gives its expression to the subtarget allocation process (i.e. assignment submodel) in each decision planning stage, where the synthesized information influences the result.

3. Algorithms and Analysis for Single-Agent Exploration
3.1. Generic Algorithm for Single-Agent Exploration
In the single-agent model (2.2.1-2.2.8), the subtarget selection in each planning stage has its effect on the overall performance, hence, it’s crucial to set some appropriate criteria for the selection. Suppose the agent is located at the $v_k$ and will select the next subtarget, say $v_{k+1}$. There are three factors that are important for the selection, they are:

1) $g(v_{k+1}, E_k) = \min_{a_0, \ldots, a_r} C(P_{v_0 \cdots v_{k+1}})$, where $P_{v_0 \cdots v_{k+1}} = v_0 \sim a_0 \sim \cdots \sim a_i \sim v_{k+1}, a_i \in E_k, i = 0, \ldots, r_k,$ which denotes the least traveling cost among all the paths from the $v_0$ to $v_{k+1}$ found by the agent up to the $k$th stage, it measures how far away $v_{k+1}$ is from the base according to the cumulative knowledge;

2) $d(v_{k+1}, t)$, which is the Euclidean distance between $v_{k+1}$ and the $t$, it heuristically measures how far away $v_{k+1}$ is from the target $t$, i.e. $d(v_{k+1}, t)$ implies the progress associated with the selection of $v_{k+1}$ in the $k$th stage;

3) $\sum_{i=0}^{k} C(P_{v_i, v_{i+1}})$, which is the agent’s cumulative traveling cost from $v_0$ to the subtarget $v_{k+1}$.

Intuitively, the smaller value of each one is preferred, this leads to an evaluator in the form:

$$f(v_{k+1}, E_k) = \lambda_1 g(v_{k+1}, E_k) + \lambda_2 d(v_{k+1}, t) + \lambda_3 \sum_{j=0}^{k} C(P_{v_j, v_{j+1}}), \quad \sum_{i=1}^{3} \lambda_i = 1, \quad \lambda_i \geq 0, \quad i = 1, 2, 3. \quad (3.1.1)$$

For the agent, in any planning stage, all the nodes can be divided into three classes: those that have
been reached, those that have not been reached but have been examined regarding the reachability, and those that have neither been reached nor examined. Obviously, during the exploration, the size of the first class is strictly monotonically increasing. The size of the second class in general does not have such monotonicity, but it’s safe to say that it is a finite number in any planning stage. The third class represents the unknown part of the world, which has an infinitely large size when the grid network is infinite. For the agent to explore, the first two classes must be recorded and updated along the exploration progress. Actually in the $k$th planning stage, the first class is $E_k$, which appears in the model (2.2.1-2.2.8), let’s denote the second class as $O_k$. Note that all the nodes in $E_k$ contribute to the map construction up to this stage; all the nodes in $O_k$ provide a candidate list for the subtarget selection in this stage. A generic algorithm, based on the operations on all the $E_k$’s and all the $O_k$’s, with the evaluator (3.1.1) for the subtarget selection, for single-agent model (2.2.1-2.2.8) is listed as the following Algorithm 3.1.1:

**Algorithm 3.1.1. Generic algorithm for single-agent exploration**

Step 1. (Initialization). Choose $\lambda_1$, $\lambda_2$, and $\lambda_3$ such that $\sum_{i=1}^{k} \lambda_i = 1$, $\lambda_i \geq 0$, $i = 1, 2, 3, k = 0$. Choose $E_k \subseteq B(s, R)$, $v_k \in E_k$. Set $O_k = \{u | u' \in \text{Neighbor}(u), u \in E_k, u' \notin E_k, a(u') < a_v\}$. For each $u \in E_k$ and each $u' \in O_k$, determine $g(u, E_k)$ and $g(u', E_k)$, let $G(u) = g(u, E_k)$ and $G(u') = g(u', E_k)$. For each $u' \in O_k$, set $\text{Pred}(u') = v_k$ (i.e. the predecessor of $u'$ is $v_k$).

Step 2. If $O_k$ is empty and $t \notin E_k$, then stop and report failure in finding target; otherwise, continue.

Step 3.

\[
\begin{align*}
\arg\min_{v \in O_k} f(v, E_k) &= \lambda_1 \cdot g(v, E_k) + \lambda_2 \cdot d(v, t) + \lambda_3 \left( \sum_{i=1}^{k-1} C(\hat{\delta}_{v_i}, \hat{\delta}_{v_i}) + C(\hat{\delta}_{v_k}, t) \right) & \text{if } k \geq 1; \\
\arg\min_{v \in O_k} f(v, E_k) &= \lambda_1 \cdot g(v, E_k) + \lambda_2 \cdot d(v, t) + \lambda_3 \cdot C(\hat{\delta}_{v_k}) & \text{if } k = 0,
\end{align*}
\]

where $\hat{\delta}_{v_k} = v_k \sim \cdots \sim \hat{\delta}_{v_i} \sim v$, $\hat{\delta}_{v_i} \in E_k$, $i = 0, \cdots, r_h$ is a least cost path extracted from $E_k$ from $v_k$ to $v$. Move to $v_{k+1}$ through the path $\hat{\delta}_{v_k, v_{k+1}}$. $E_{k+1} = E_k \cup \{v_{k+1}\}$. If $t \in E_{k+1}$, then stop and report success in finding target, determine $g(t, E_{k+1})$ by the A* algorithm; otherwise, continue.

Step 4. For each $v \in \text{Neighbor}(v_{k+1})$, $v \notin E_{k+1}$ evaluate $a(v)$. If $a(v) \geq a_v$, ignore $v$; if $a(v) < a_v$ and $v \notin O_k$, then $O_{k+1} = (O_k \setminus \{v_{k+1}\}) \cup \{v\}$, determine $g(v, E_{k+1})$, let $G(v) = g(v, E_{k+1})$, and set $\text{Pred}(v) = v_{k+1}$; if $a(v) < a_v$ and $v \in O_k$, then $O_{k+1} = O_k - \{v_{k+1}\}$, determine $g(v, E_{k+1})$, let $G(v) = g(v, E_{k+1})$ and $\text{Pred}(v) = v_{k+1}$. $k = k + 1$. Go to step 2.

During the exploration, the agent dynamically stores and updates two node lists, i.e. $E_k$ and $O_k$ in the $k$th planning stage are updated into $E_{k+1}$ and $O_{k+1}$ in the $(k+1)$th planning stage. There are only two situations for the agent to stop. The first situation is for some $k$, $O_k$ is empty and $t \notin E_k$. In this case, the agent reports failure in finding the target $t$. It can be shown that this type of report implies that
there is no path from the base to the target. The second situation is \( t \in E_k \) for some \( k \). In this case, the agent reports success in finding the target \( t \). It can also be shown that as long as there exists a path from the base to the target, the agent can find the target within finite steps of moves. Also during the exploration, each time the agent examines a node as a neighbor of its location, a label, say \( G \), is attached to this node. This \( G \) label stands for the “length” of a “shortest” path in the current map from the starting node \( v_0 \) to this node. The \( G \) label can be updated when this node is reexamined and a “short cut” resulting from the map increment is found. The \( G \) label will remain unchanged if this node is reached. The \( G \) label for the target \( t \) stands for the length of a shortest path in the final map from the starting node \( v_0 \) to \( t \). If \( t \) is found, the \( G \) label for \( t \) is a measure of the agent’s merit. For each node with a \( G \) label, there is another label, say \( \text{Pred} \), that indicates the predecessor of the node in a shortest path in the current map. When the map reaches its final version and \( t \) is found, then by a finite sequence of predecessors, a shortest path in the final map can be constructed. Those claims are based on direct observations; next, in this section, we study the properties of the Algorithm 3.1.1 in a more rigorous way. We will also find a practical way of implementation.

**Definition 3.1.1.** In any planning stage, the explored region is the set of all the reached nodes in this stage.

**Remark.** It is obvious that in the \( k \)th planning stage, the explored region is \( E_k \). It’s reasonable to assume that \( E_0 \), which represents a set of reachable nodes in base, are pairwise connected. When we say a region is connected, it is for any two nodes in this region, there exists a path in this region that connects them. Since \( E_{k+1} = E_k \cup \{v_{k+1}\} \) and the agent reaches \( v_{k+1} \) through a least cost path in the map associated with \( E_k \), it suffices that the node \( v_{k+1} \) is connected with some node in \( E_k \), hence if \( E_k \) is connected then \( E_{k+1} \) is connected too. This leads to the following theorem.

**Theorem 3.1.1.** In any planning stage, the explored region is connected.

**Definition 3.1.2.** In any planning stage, the frontier associated with the explored region is a set that consists of all the reachable yet unreached outer boundary nodes of the explored region.

**Theorem 3.1.2.** In the \( k \)th planning stage, the frontier is \( O_k \).

**Proof.** Consider all the outer boundary nodes of \( E_k \). They can be divided into two classes: those that are reachable and those that are not. By the Definition 3.1.2, the frontier, say \( F_k \), is the first class. Note that \( O_k = \{u' | u' \in \text{Neighbor}(u), u \in E_k, u' \notin E_k, a(u') < a_c\} \) implies \( O_k \subseteq F_k \). Since for any node \( u' \in F_k \), we have \( u' \notin E_k, a(u') < a_c \), and there must be a node \( u \in E_k \) such that \( u \in \text{Neighbor}(u') \), i.e. \( u' \in \text{Neighbor}(u) \), hence \( u' \in O_k \), which implies \( F_k \subseteq O_k \). \( \square \)

**Remark.** This theorem implies that after the agent reaches a node, the frontier is extended (or “pushed”); the exploded region is expanded. Once the target node \( t \) appears in the frontier, a path from the base to \( t \) is in fact already found. If \( t \) enters some \( E_k \), then a path of minimal traveling cost from \( v_0 \) to \( t \) is found. This is what the following theorem says.

**Definition 3.1.3.** If there exists a \( N \in \mathbb{Z}^+ \), \( N > 0 \) such that \( t \in E_N \), then \( G(t) \) is said to be *conditionally*
minimum if it equals the least traveling cost among all the paths from \( v_0 \) to \( t \) retrieved from \( E_N \).

Theorem 3.1.3. If there exists a \( N \in \mathbb{Z}^1, N > 0 \) such that \( t \in E_N \), then \( G(t) \) is conditionally minimum.

Proof. By Step 3 and Step 4 in the Algorithm 3.1.1, we have that for each \( v_{k+1} \) such that \( \bar{E}_k \cup G(v_{k+1}) = g(v_{k+1}, E_k) \), which implies \( G(t) = g(t, E_{N+1}) \) if \( t \in E_N \) for some \( N > 0 \). Note that \( E_N \) is maximal, we have that \( g(t, E_{N+1}) = g(t, E_N) \), which implies \( g(t, E_N) \), i.e. \( G(t) \) is conditionally minimum.

Lemma 3.1.1. If for some \( k \in \mathbb{Z}^1, k > 0 \), \( O_k \) is empty and \( t \not\in E_k \), then there is no path from the base to the target \( t \).

Proof. Suppose (for contradiction) that there exists a path, say \( \gamma \), from \( v_0 \) to \( t \). Let \( \alpha \) be the last node in \( \gamma \) such that \( \alpha \in \bigcup_{i=0}^{k} O_i \). Note that such \( \alpha \) must exists because \( v_0 \in E_0 \). We can claim that \( \alpha \neq t \), otherwise \( t \in \bigcup_{i=0}^{k} O_i \) implies \( t \in O_i \) for some \( i < k \). But \( O_k \) is empty, hence there must be a \( j, i < j \leq k \) such that \( t \in E_j \). Note that \( E_j \subseteq E_k \), hence \( t \in E_k \).

Since \( \alpha \neq t \), there must be a node, say \( \beta \), such that \( \beta \) is the successor of \( \alpha \) in \( \gamma \). Since \( \alpha \in \bigcup_{i=0}^{k} O_i \), we must have that \( \alpha \in O_i \) for some \( i < k \). But \( O_k \) is empty, hence there must be a \( j, i < j \leq k \) such that \( \alpha \in E_j \). Note that \( \beta \in \text{Neighbor}(\alpha) \), hence by Step 4 in the Algorithm 3.1.1, we have that \( \beta \in O_j \) for some \( i' \leq j \). But this implies that \( \beta \in \bigcup_{i=0}^{k} O_i \), hence \( \alpha \) is not the last one that is in this union, a contradiction!

Remark. This lemma is very meaningful. It tells that once the frontier vanishes, there is no reason for the agent to move on because there is no path to the target \( t \). Conversely, it is not true in general. Imagine there is an infinitely long “wall” that separates the base and the target, then it is possible that \( O_k \) is not empty for any \( k > 0 \). In this extreme case, the agent will not stop forever. A practical condition is that the base is inside a region enclosed by a finitely long closed “wall”, which separates the base and the target. Of course, there is no path from the base to the target in this case. The following lemma says that the agent will be able to verify it within finite steps of decision-making and moves.

Lemma 3.1.2. If there is a connected region that contains finite number of nodes, and for any node \( v \) in the outer boundary of the region, \( a(v) \geq a_{\infty} \), and the base is in side this region; the target \( t \) is outside this region, then there is a \( k \in \mathbb{Z}^1, k > 0 \) such that \( O_k \) is empty and \( t \not\in E_k \), \( i = 0, 1, \ldots, k \).

Proof. Since the outer boundary of the region consists of all the node \( v \)’s such that \( a(v) \geq a_{\infty} \), then there is no path from the base to the target \( t \), hence \( t \not\in E_i \) for any \( i \geq 0 \). Suppose (for contradiction) that for any \( k > 0 \), \( O_k \) is not empty, then by Step 3 in the Algorithm 3.1.1, the set sequence \( \{E_k\} \) is strictly monotonically increasing. However, the size of \( E_k \), say \( |E_k| \), is bounded above, hence there must be a \( k' > 0 \) such that \( E_{k+1} = E_{k'} \). This implies that \( O_{k'} \) is empty, hence a contradiction.

Remark. Lemma 3.1.1 and Lemma 3.1.2 give the indicator of failing exploration. It’s obvious that \( t \in E_k \) for some \( k > 0 \) is an indicator of successful exploration. A question is if there exists a path from the
base to the target \( t \), can the agent find it within finite steps of decision-making and moves? The following theorem, as a sufficient condition for the completeness of single-agent exploration in grid worlds, gives the answer.

Theorem 3.1.4. If there exists a path from the base to the target \( t \) and \( \lambda_3 = 0 \), then there must be a \( N \in \mathbb{Z}^+, N > 0 \), such that \( t \in E_N \).

Proof. When \( \lambda_3 = 0 \), we have \( \lambda_1 + \lambda_2 = 1, \lambda_1 \geq 0, \lambda_2 \geq 0 \). Suppose (for contradiction) that for any \( k > 0 \), \( t \notin E_k \). Since there exists a path from the base to the target \( t \), by the Lemma 3.1.1, we have that \( O_k \) is not empty for any \( k > 0 \), hence, by Step 3 in the Algorithm 3.1.1, the set sequence \( \{ E_k \} \) strictly monotonically increases without upper bound. Suppose \( \gamma \) is a path from \( v_0 \) to \( t \) and \( \alpha \) is the last node in \( \gamma \) such that \( \alpha \in E_i \) for some \( i > 0 \). Note that \( v_0 \in E_0 \), hence such \( \alpha \) must exist. Since \( t \notin E_k \) for any \( k > 0 \), we have that \( \alpha \) must have a successor, say \( \beta \), in \( \gamma \).

Since \( \alpha \in E_i \), by Step 4 in the Algorithm 3.1.1, there must be a first \( j \), \( 0 < j \leq i \), such that \( \beta \notin E_j \). Since \( \beta \notin E_j \) for any \( k > 0 \) and \( \beta \in O_i \), we have that \( \beta \in O_k \) for any \( k \geq j \).

Note that \( g(\beta, E_j) \) exists and is finite. Consider the node set \( \psi = \{ v \mid d(v, t) \leq d(\beta, t) \) or \( d(v_0, v) \leq g(\beta, E_j) \} \). Since both \( d(\beta, t) \) and \( g(\beta, E_j) \) are finite and the grid size \( \delta > 0 \) is fixed, we have that there are only finite number of nodes in \( \psi \). Denote \( \psi^c = \{ v \mid d(v, t) > d(\beta, t) \) and \( d(v_0, v) > g(\beta, E_j) \} \). Note that the sequence \( \{|E_k|\} \) is unbounded above, hence, the sequence \( \{|E_k \cap \psi^c|\} \) is unbounded above.

Let \( j \) be sufficiently large such that \( \beta \in O_j \) and \( E_j \cap \psi^c \) is not empty. Moreover, there must exists a \( u \in \psi^c \) such that \( u \notin E_j \) and \( E_j \cup \{ u \} = E_{j+1} \), then \( u \in O_j \). Note that \( u \neq \beta \).

Since \( u \in \psi^c \), we have that \( \lambda_1 v(u, E_j) + \lambda_2 d(u, t) \geq \lambda_1 v(v_0, u) + \lambda_2 d(u, t) > \lambda_1 g(\beta, E_j) + \lambda_2 d(\beta, t) \).

The second inequality is strict because at least one of \( \lambda_1 \) and \( \lambda_2 \) is positive. Hence \( f(u, E_j) > f(\beta, E_j) \).

Note that \( j \geq j \), hence \( g(\beta, E_j) \geq g(\beta, E_j) \), which implies \( f(u, E_j) \geq f(\beta, E_j) \), hence \( f(u, E_j) > f(\beta, E_j) \). But this violates the selection rule in the Step 3 in the Algorithm 3.1.1 because both \( u \) and \( \beta \) are in \( O_j \). \( \square \)

Remark. This theorem gives a strong sufficient condition for the completeness of the exploration. That is, if a path from the base to the target \( t \) exists, then the agent can find \( t \) within finite moves. This condition drops the third term in the evaluator (3.1.1). However, we do care the cumulative traveling cost for the agent to move to \( t \). In the following studies, we will disclose that the cumulative traveling cost in fact could be affected by the values of \( \lambda_1 \) and \( \lambda_2 \). Since \( \lambda_3 = 0 \) implies \( \lambda_1 + \lambda_2 = 1 \), we can set \( \epsilon = \lambda_1 \) and then \( \lambda_2 = 1-\epsilon \). The value of \( \epsilon \) could be any number between 0 and 1, where the completeness property holds. The next question is on the efficiency and the effectiveness, which are measured by the objective values in (2.2.1).

3.2. Practical Implementation for Single-Agent Exploration

In the case that the target is geographically far away from the base, the size of \( E_k \) could be very large if \( k \) is large. Hence, for any \( v \in O_k \), determining \( G(v) = g(v, E_k) \) is computationally expensive. Also, it’s possible that the size of \( O_k \) is considerably large, hence the direct sweeping of the labels attached to all the nodes in \( O_k \) is not economical. To avoid the computational cost in evaluating such \( G(v) \), the value of \( G(v') \) could be used if it already exists, where \( v' \in Neighbor(v) \), hence, \( G(v) \) is set when \( v \) has a reached predecessor; \( G(v) \) is improved when a “better” reached predecessor is determined. In this way the \( G \) label for any reached node can be quickly determined at the expense of the conditional
Step 1. (Initialization). Choose \( \varepsilon \), \( 0 \leq \varepsilon \leq 1 \). \( k = 0 \). Choose \( E_k \subseteq B(s, R) \), \( v_k \in E_k \). Set \( O_k = \{ u \mid u \in \text{Neighbor}(u), u \in E_k, u \notin E_k \} \). Store \( O_k \) as \( bi_{heap} \). For each \( u \in E_k \) and each \( u' \in O_k \), determine \( g(u, E_k) \) and \( g(u', E_k) \), let \( G(u) = g(u, E_k) \) and \( G(u') = g(u', E_k) \). For each \( u' \in O_k \), set \( \text{Pred}(u') = v_k \) (i.e., the predecessor of \( u' \) is \( v_k \)).

Step 2. If \( O_k \) is empty and \( t \notin E_k \), then stop and report failure in finding target; otherwise, continue.

Step 3. Find \( v_{k+1} = \arg \min_{v \in O_k} \{ G(v) + (1-\varepsilon) \cdot d(v, t) \} \) as the top element in \( bi_{heap} \) denote \( \hat{P}_{v_k,v} = \)= \( v_k \sim \hat{\alpha}_0 \sim \cdots \sim \hat{\alpha}_i \sim v \), \( \hat{\alpha}_i \in E_k \), \( i = 0, \cdots, r_k \) as a least cost path extracted from \( E_k \) from \( v_k \) to \( v \).

Move to \( v_{k+1} \) through the path \( \hat{P}_{v_k,v_{k+1}} \). \( E_{k+1} = E_k \cup \{ v_{k+1} \} \). If \( t \in E_{k+1} \), then stop and report success in finding target, determine \( g(t, E_{k+1}) \) by the A* algorithm; otherwise, continue.

Step 4. \( bi_{heap}^{k+1} = bi_{heap}^k \). For each \( v \in \text{Neighbor}(v_{k+1}), v \notin E_{k+1} \) evaluate \( a(v) \). If \( a(v) \geq a_\alpha \), ignore \( v \); if \( a(v) < a_\alpha \) and \( v \notin O_k \), then \( O_{k+1} = (O_k \setminus \{ v_{k+1} \}) \cup \{ v \} \), update \( bi_{heap}^{k+1} \) by removing \( v_{k+1} \) and entering \( v \), let \( G(v) = G(v_{k+1}) + C(v_{k+1}, v) \), and set \( \text{Pred}(v) = v_{k+1} \); if \( a(v) < a_\alpha \) and \( v \in O_k \), then \( O_{k+1} = O_k \setminus \{ v_{k+1} \} \), let \( G(v) = G(v_{k+1}) + C(v_{k+1}, v) \) and set \( \text{Pred}(v) = v_{k+1} \) if \( G(v) > G(v_{k+1}) + C(v_{k+1}, v) \), update \( bi_{heap}^{k+1} \) by repairing the current \( bi_{heap}^{k+1} \). \( k = k + 1 \). Go to step 2.

It’s easy to verify that under the commands of the Algorithm 3.2.1, the Theorem 3.1.1 and the Theorem 3.1.2 still hold. The Theorem 3.1.3 does not hold anymore. We will study the relation between the value of \( G(t) \) and \( \varepsilon \) thereafter. It’s also easy to verify that the Lemma 3.1.1 and the lemma 3.1.2 still hold. With slight modification to the proof for the Theorem 3.1.4, we can also verify that it still holds for the Algorithm 3.2.1, that is:

Theorem 3.2.1. Under the commands of the Algorithm 3.2.1, if there exists a path from the base to the target \( t \), then there must be a \( N \in \mathbb{Z}^+ \), \( N > 0 \), such that \( t \in E_N \).

Proof. Suppose (for contradiction) that for any \( k > 0 \), \( t \notin E_k \). Since there exists a path from the base to the target \( t \), the Lemma 3.1.1 implies that the set sequence \( \{ E_k \} \) strictly monotonically increases without upper bound. Suppose \( \gamma \) is a path from \( v_0 \) to \( t \) and \( \alpha \) is the last node in \( \gamma \) such that \( \alpha \in E_i \) for some \( i > 0 \). We know that \( \alpha \) must have a successor, say \( \beta \), in \( \gamma \) and there must be a first \( j \), \( 0 < j \leq i \), such that \( \beta \in O_j \). Since \( \beta \notin E_k \) for any \( k > 0 \) and \( \beta \in O_j \), we have that \( \beta \in O_k \) for any \( k \geq j \). Let \( G^{(0)}(\beta) \) be the \( G \) label for \( \beta \) when \( \beta \) enters \( O_j \). Consider the node set \( \psi = \{ v \mid d(v, t) \leq d(\beta, t) \text{ or } d(v_0, v) \leq G^{(0)}(\beta) \} \). Since both \( d(\beta, t) \) and \( G^{(0)}(\beta) \) are finite and the grid size \( \delta > 0 \) is fixed, we have that there are
only finite number of nodes in $\psi$. Denote $\psi^c = \{v \mid d(v, t) > d(\beta, t) \text{ and } d(v_0, v) > G(0)(\beta)\}$. Note that the sequence $\{|E_\beta|\}$ is unbounded above, the sequence $\{|E_\beta \cap \psi^c|\}$ is also unbounded above. There must be a sufficiently large $j'$ such that $\beta \in O_j$ and $E_{j'} \cap \psi^c$ is not empty. Moreover, there must exists an $u \in \psi^c$ such that $u \not\in E_{j'}$ and $E_{j'} \cup \{u\} = E_{j'+1}$, then $u \not\in O_{j'}$. Note that $u \not= \beta$. Let $G^{(1)}(u)$ be the $G$ label for $u$ when $u$ is in $O_{j'}$ and about to enter $E_{j'+1}$; let $G^{(1)}(\beta)$ be the $G$ label for $\beta$ at the same moment. Since $u \in \psi^c$, we have that $\varepsilon G^{(1)}(u) + (1-\varepsilon)\cdot d(u, t) \geq \varepsilon d(v_0, u) + (1-\varepsilon)\cdot d(u, t) > \varepsilon G^{(0)}(\beta) + (1-\varepsilon)\cdot d(\beta, t)$. Note that $j' \geq j$, hence $G^{(0)}(\beta) \geq G^{(1)}(\beta)$, which implies $\varepsilon G^{(1)}(u) + (1-\varepsilon)\cdot d(u, t) > \varepsilon G^{(1)}(\beta) + (1-\varepsilon)\cdot d(\beta, t)$. But this violates the selection rule in the Step 3 in the Algorithm 3.2.1 because both $u$ and $\beta$ are in $O_{j'}$. □

Remark. The successful shift from the proof for the Theorem 3.1.4 to the Theorem 3.2.1 lies in the fact that in the Algorithm 3.2.1 the $G$ label for any node $v$, say $G(v)$, if exists, is still nonincreasing. It decreases if $\text{pred}(v)$ is changed (in this case, we say an improvement is made). It remains unaltered if $v$ enters $E_k$ for some $k > 0$. An improvement of $G(v)$ may contribute to the less value of $G(t)$ if it can exist. We now turn to study the value of $G(t)$. The following results in this section are all from the Algorithm 3.2.1.

Definition 3.2.1. If there exists a $N \in \mathbb{Z}^+$, $N > 0$ such that $t \in E_N$, then $G(t)$ is said to be absolutely minimum if it equals the least traveling cost among all the paths from $v_0$ to $t$. Also, for any node $v \in E_k$ for some $k \in \mathbb{Z}^+$, $0 \leq k \leq N$, $G(v)$ is absolutely minimum if it equals the least traveling cost among all the paths from $v_0$ to $v$.

Lemma 3.2.1. If there exists a $N \in \mathbb{Z}^+$, $N > 0$ such that $t \in E_N$ and $P_{v_0,t}$ is a path from $v_0$ to $t$, then for any $k \in \mathbb{Z}^+$, $0 \leq k < N$, there exists a node, say $\beta$, in $P_{v_0,t}$ such that $\beta \in O_k$ and each node before $\beta$ in $P_{v_0,t}$ is in $E_k$.

Proof. Note that $t \in E_N$ and $t \not\in E_k$ for any $k < N$, hence, for any $k < N$, the set $\theta = \{v \mid v \not\in P_{v_0,t} \text{ and } v \in O_k\}$ is not empty. In fact $\theta$ could be viewed as the intersection between the node set of $P_{v_0,t}$ and the frontier of the $k$th planning stage (i.e. $O_k$). Let $\beta \in \theta$ be the node that the number of edges from $v_0$ to $\beta$ in $P_{v_0,t}$ is the smallest. Hence, each node $\alpha$ before $\beta$ in $P_{v_0,t}$ is in $E_k$. □

Remark. Let’s denote such $\beta$ as $s(P_{v_0,t}, O_k)$ for the preparation of the following lemma.

Lemma 3.2.2. If there exists a $N \in \mathbb{Z}^+$, $N > 0$ such that $t \in E_N$ and $P_{v_0,t}^*$ is a least cost path from $v_0$ to $t$, then for any $k \in \mathbb{Z}^+$, $0 \leq k < N$ and each node $\alpha$ before $s(P_{v_0,t}^*, O_k)$ in $P_{v_0,t}^*$, $G(\alpha)$ exists and is absolutely minimum.
Proof. By the Lemma 3.2.1, all the nodes before \( s(P_{v_0,t}^*, O_k) \) in \( P_{v_0,t}^* \) are in \( E_k \). Let \( \alpha_0 (= v_0) \sim \alpha_1 \sim \cdots \sim \alpha_j (j \geq 1) \) be the subpath of \( P_{v_0,t}^* \) formed by all those nodes. Note that for each \( i = 0, 1, \cdots, j \), \( G(\alpha_i) \) exists and unaltered. Moreover, \( G(\alpha_i) \leq G(\alpha_{i+1}) + C(\alpha_{i+1}, \alpha_i) \), \( \cdots \), \( G(\alpha_i) \leq G(\alpha_0) + C(\alpha_0, \alpha_1) \). By adding some of these inequalities together and note that \( G(\alpha_0) = G(v_0) = 0 \), we have that \( G(\alpha_i) \leq \sum_{q=0}^{i-1} C(\alpha_q, \alpha_{q+1}) = C(P_{v_0,\alpha_i}^*) \) for each \( i = 1, \cdots, j \), where \( P_{v_0,\alpha_i}^* \) denotes the least cost subpath from \( v_0 \) to \( \alpha_i \). However, \( G(\alpha_i) \geq C(P_{v_0,\alpha_i}^*) \) for each \( i = 1, \cdots, j \). Hence, \( G(\alpha_i) = C(P_{v_0,\alpha_i}^*) \) for each \( i = 1, \cdots, j \), i.e. they are absolute minimum. \( \square \)

Lemma 3.2.3. If \( \frac{1}{2} \leq \varepsilon \leq 1 \), and there exists a \( N \in \mathbb{Z}^+ \), \( N > 0 \) such that \( t \in E_N \), then for any \( k \in \mathbb{Z}^+ \), \( 0 \leq k < N \), there is a node \( \beta \in O_k \) such that \( f(\beta) \leq \varepsilon \cdot C(P_{v_0,\beta}^*) \), where \( P_{v_0,\beta}^* \) is a least cost path from \( v_0 \) to \( \beta \).

Proof. By the Lemma 3.2.1, there exists a node, say \( \beta \), in \( P_{v_0,t}^* \) such that \( \beta \in O_k \) and each node before \( \beta \) in \( P_{v_0,t}^* \) is in \( E_k \). Let \( \alpha \) be the node immediately before \( \beta \) in \( P_{v_0,t}^* \), hence \( \alpha \in E_k \). By the Lemma 3.2.2, \( G(\alpha) \) exists and is absolutely minimum. Since \( \beta \in \text{Neighbor}(\alpha) \), we have that \( G(\beta) \) exists and \( G(\beta) \leq G(\alpha) + C(\alpha, \beta) \). Hence \( f(\beta) = \varepsilon \cdot G(\beta) + (1-\varepsilon) \cdot d(\beta, t) = \varepsilon \cdot \left( G(\beta) + \frac{1-\varepsilon}{\varepsilon} \cdot d(\beta, t) \right) \leq \varepsilon \cdot G(\beta) + d(\beta, t) \). Note that \( G(\alpha) + C(\alpha, \beta) = C(P_{v_0,\beta}^*) \), where \( P_{v_0,\beta}^* \) denotes the least cost subpath from \( v_0 \) to \( \beta \). But \( G(\beta) \geq C(P_{v_0,\beta}^*) \). Hence \( G(\beta) = C(P_{v_0,\beta}^*) \). Note that \( d(\beta, t) \leq C(P_{\beta,t}^*) \), where \( P_{\beta,t}^* \) denotes the least cost subpath from \( \beta \) to \( t \), hence \( \varepsilon \cdot (G(\beta) + d(\beta, t)) \leq \varepsilon \cdot (C(P_{v_0,\beta}^*) + C(P_{\beta,t}^*)) = \varepsilon \cdot C(P_{v_0,t}^*) \), which implies that \( f(\beta) \leq \varepsilon \cdot C(P_{v_0,t}^*) \). \( \square \)

Remark. This lemma is a generalized version of one of the Nilsson’s results for A* Algorithm [31] that fits for the grid worlds. It directly leads to a sufficient optimality condition of \( G(t) \), which is summarized as the following theorem.

Theorem 3.2.1. If \( \frac{1}{2} \leq \varepsilon \leq 1 \), and there exists a \( N \in \mathbb{Z}^+ \), \( N > 0 \) such that \( t \in E_N \), then \( G(t) \) is absolutely minimum when \( t \in E_N \).

Proof. Suppose (for contradiction) that \( G(t) \) is not absolutely minimum. Then, when \( t \) enters \( E_N \), for
any least cost path \( P_{v_0,t}^* \) from \( v_0 \) to \( t \), \( G(t) > C(P_{v_0,t}^*) \), By the nonincreasing property of the \( G \) label for \( t \), when \( t \) is in \( O_{N-1} \), \( G(t) > C(P_{v_0,t}^*) \) holds too. But this implies that \( f(t) = \epsilon \cdot G(t) + (1-\epsilon) \cdot d(t, t) = \epsilon \cdot G(t) > \epsilon \cdot C(P_{v_0,t}^*) \) when \( t \in O_{N-1} \). By the selection rule in the Step 3 in the Algorithm 3.2.1, we have that for any \( v \in O_{N-1} \), \( f(v) \geq \epsilon \cdot C(P_{v_0,t}^*) \), which, however, contradicts the Lemma 3.2.3.

Remark. The theorem tells that \( \frac{1}{2} \) is a critical value, as long as \( \epsilon \geq \frac{1}{2} \), it is guaranteed that the agent, under the commands of the Algorithm 3.2.1, is be able to find an absolute least cost path from \( v_0 \) to \( t \). Since there are infinitely many numbers between \( \frac{1}{2} \) and 1, a natural question is which one to choose? The following lemmas and theorem, which are generalized version of some Nilsson’s results for A* Algorithm [31], give some hints to this question.

Lemma 3.2.4. If \( \frac{1}{2} \leq \epsilon \leq 1 \), and there exists a \( N \in \mathbb{Z}^1, N > 0 \) such that \( t \in E_N \), then for any node \( v \in E_N \), \( f(v) \leq \epsilon \cdot C(P_{v_0,t}^*) \) at the time \( t \in E_N \), where \( P_{v_0,t}^* \) is a least cost path from \( v_0 \) to \( t \).

Proof. Suppose (for contradiction) that there is a node \( v \in E_N \) such that \( f(v) > \epsilon \cdot C(P_{v_0,t}^*) \) at the time \( t \in E_N \). Since \( v \in E_N \), there must be a \( k, 0 \leq k < N \), such that \( v \in O_k \) and \( v \in E_{k+1} \). Note that the nonincreasing property of the \( G \) label for \( v \) implies the nonincreasing property of the \( f \) label for \( v \), hence when \( v \in O_k, f(v) > \epsilon \cdot C(P_{v_0,t}^*) \) also holds. But \( v \) is selected to enter \( E_{k+1} \), hence by the Step 3 in the Algorithm 3.2.1, in the \( k \)th planning stage, for any \( u \in O_k \), we have that \( f(u) \geq f(v) > \epsilon \cdot C(P_{v_0,t}^*) \), which contradicts the Lemma 3.2.3.

Lemma 3.2.5. If \( \frac{1}{2} \leq \epsilon \leq 1 \), and there exists a \( N \in \mathbb{Z}^1, N > 0 \) such that \( t \in E_N \), then for any node \( v \) that has a \( f \) label such that \( f(v) < \epsilon \cdot C(P_{v_0,t}^*) \) at some time, it must hold that \( v \in E_N \), where \( P_{v_0,t}^* \) is a least cost path from \( v_0 \) to \( t \).

Proof. Suppose (for contradiction) that \( v \not\in E_N \), then \( v \in O_{N-1} \). By the nonincreasing property of the \( f \) label for \( v, f(v) < \epsilon \cdot C(P_{v_0,t}^*) \) also holds when \( v, t \in O_{N-1} \) and \( t \) is about to enter \( E_N \). By Theorem 3.2.1, \( f(t) = \epsilon \cdot C(P_{v_0,t}^*) \) when \( t \in O_{N-1} \) and about to enter \( E_N \). Hence by the end of the \((N-1)\)th planning stage,
\( f(v) < f(t) = \varepsilon \cdot C(P_{v,t}^*) \), which contradicts the selection rule in the Step 3 in the Algorithm 3.2.1. □

Theorem 3.2.2. If \( \varepsilon_1 \in R^1, \varepsilon_2 \in R^1, \frac{1}{2} \leq \varepsilon_1 < \varepsilon_2 \leq 1 \), and there exists a \( N_1 \in Z^l \), \( N_1 > 0 \) such that \( t \in E_{N_1} \) when \( \varepsilon = \varepsilon_1 \); there exists a \( N_2 \in Z^l, N_2 > 0 \) such that \( t \in E_{N_2} \) when \( \varepsilon = \varepsilon_2 \), then \( E_{N_1} \subseteq E_{N_2} \).

Proof. Let \( f_{\varepsilon_1}(u) \) denote the \( f \) label of \( u \) (if it exists) when \( \varepsilon = \varepsilon_1, t \in E_{N_1} \); let \( f_{\varepsilon_2}(v) \) denote the \( f \) label of \( v \) (if it exists) when \( \varepsilon = \varepsilon_2, t \in E_{N_2} \). If \( u \in E_{N_1} \), \( u \neq t \), then \( f_{\varepsilon_1}(u) \) exists. Note that when \( t \in E_{N_1} \), \( f_{\varepsilon_1}(u) = \varepsilon_1 \cdot G_{\varepsilon_1}(u) + (1-\varepsilon) d(u, t) \) is unaltered and \( G_{\varepsilon_1}(u) \) represents the traveling cost of some path \( P_{v_0,u}^* \) from \( v_0 \) to \( u \). Note also that in \( P_{v_0,u}^* \) each node, say \( \alpha \), is the predecessor of the node immediately after \( \alpha \), hence the path \( P_{v_0,u}^* \) can be viewed as a pointer path from \( v_0 \) to \( u \). Since each node, say \( \alpha \), in \( P_{v_0,u}^* \) is in \( E_{N_1} \), by the Lemma 3.2.4, we must have that when \( t \in E_{N_1} \), \( f_{\varepsilon_1}(\alpha) \) exists and \( f_{\varepsilon_1}(\alpha) \leq \varepsilon_1 \cdot C(P_{v_0,u}^*) \), where \( P_{v_0,u}^* \) is a least cost path from \( v_0 \) to \( t \).

Now, consider \( \varepsilon = \varepsilon_2, t \in E_{N_2} \). Suppose (for contradiction) \( u \notin E_{N_2} \), then the intersection between the node set of \( P_{v_0,u}^* \) and \( O_{N_2-1} \) is not empty. Let \( v \) be the node in this intersection such that the subpath of \( P_{v_0,u}^* \) from \( v_0 \) to \( v \) has the smallest number of edges, hence each node before \( v \) in \( P_{v_0,u}^* \) is in \( E_{N_2-1} \), which implies that \( G_{\varepsilon_2}(v) \leq G_{\varepsilon_1}(v) \), hence, \( G_{\varepsilon_2}(v) + \frac{1-\varepsilon_2}{\varepsilon_2} d(v,t) < G_{\varepsilon_1}(v) + \frac{1-\varepsilon_1}{\varepsilon_1} d(v,t) \).

Since \( v \) is a node of \( P_{v_0,u}^* \), by previous result, \( f_{\varepsilon_1}(v) \leq \varepsilon_1 \cdot C(P_{v_0,u}^*) \), i.e. \( G_{\varepsilon_1}(v) + \frac{1-\varepsilon_1}{\varepsilon_1} d(v,t) \leq C(P_{v_0,u}^*) \), hence \( G_{\varepsilon_2}(v) + \frac{1-\varepsilon_2}{\varepsilon_2} d(v,t) < C(P_{v_0,u}^*) \), i.e. \( f_{\varepsilon_2}(v) < \varepsilon_2 \cdot C(P_{v_0,u}^*) \). By the Lemma 3.2.5, we have \( v \in E_{N_2} \). But this contradicts the fact that \( t \in O_{N_2-1} \) and \( t \in E_{N_2} \) since \( v \neq t \). □

Conjecture 3.2.1. If \( \varepsilon_1 \in R^1, \varepsilon_2 \in R^1, \frac{1}{2} \leq \varepsilon_1 < \varepsilon_2 \leq 1 \), there exists a \( N_1 \in Z^l \), \( N_1 > 0 \) such that \( t \)
\(\in E_{N_{1}}\) when \(\varepsilon = \varepsilon_{1}\); there exists a \(N_{2}\in \mathbb{Z}, N_{2} > 0\) such that \(t \in E_{N_{2}}\) when \(\varepsilon = \varepsilon_{2}\), and if the agent always travels along a least cost path from its location to its next subtarget, then the total traveling cost associated with \(\varepsilon = \varepsilon_{1}\) is no greater than that associated with \(\varepsilon = \varepsilon_{2}\).

Remark 1. Intuitively, By the Theorem 3.2.2 we can see that every node that is reached under \(\varepsilon = \varepsilon_{1}\) is also reached under \(\varepsilon = \varepsilon_{2}\). Obviously, if the subtarget sequence associated with \(\varepsilon = \varepsilon_{1}\) is a subsequence of that associated with \(\varepsilon = \varepsilon_{2}\), then the conclusion is straightforward by the triangle inequality. However, a lot of numerical experiments show that this assumption is not always true although the conjecture is always true.

Remark 2. A lot of numerical experiments also show that although the agent takes a confined path (i.e. the nodes of the path are restricted to be those already reached) to reach its next subtarget from its current location, the cost of this confined path is usually absolute minimum. One reason is that the subtarget is usually very close to the agent’s current location. It may make sense to choose \(\varepsilon = \frac{1}{2}\) in Step 3 in the Algorithm 3.2.1 since \(\varepsilon = \frac{1}{2}\) seems to represent the least exploration effort while guarantying an absolute minimum cost path found finally. However, from the practical point of view, to find a least cost path from \(v_{0}\) to \(t\) is not as important as to find the target \(t\) swiftly, and a suboptimal path is usually acceptable. In the later sections of this paper, some listed numerical results show that the exploration effort of a single agent is still considerably large when \(\varepsilon = \frac{1}{2}\). And this leads to the slowness of target finding. An idea is to reduce the value of \(\varepsilon\) further. Note that the completeness of the exploration still holds.

4. Algorithms and Analysis for Multi-Agent Exploration

4.1. A Dynamic Assignment Model

As claimed in literature \([44, 45, 50, 51]\), multiple agents are more fault-tolerant than a single agent. If there are \(m > 1\) agents, at any time, say \(k\), each of them has an independent failure probability, say \(p_{k}^{(j)}\), then the probability that all the agents fail would be \(\prod_{j=1}^{m} p_{k}^{(j)}\). Also, if multiple agents explore the unknown world concurrently, then after each planning stage, the knowledge on the world increases much faster than that acquired by a single agent. This is indicated by (2.2.11) in the multi-agent model (2.2.9-2.2.16). Based on the model (2.2.9-2.2.16), a synchronized multi-agent exploration algorithm can be built in this manner: 1) consider the target finding based on team exploration as a multi-stage decision planning process; 2) in each stage, if possible, determine a subtarget for each agent such that no subtarget is assigned to two different agents; 3) after each agent reaches its subtarget, the central map is updated with the explored region expanded and the frontier pushed, and, at the same time, with the new node labels attached. Obviously, in each planning stage, as a submodel, an assignment problem must be solved quickly. In this section, we study a generic assignment model and then apply the result to the multi-agent exploration in the following sections.
The elements of the assignment model are as follows. There are two finite and disjoint sets $A$ and $T$: $A = \{A_1, A_2, \cdots, A_m\}$ is the set of agents, $T = \{t_1, t_2, \cdots, t_n\}$ is the set of subtargets, and $n \geq m$. Each agent $A_j$, $j = 1, 2, \cdots, m$ has a strict preference over all the subtargets in $T$. This preference is denoted as an ordered list $P(A_j) = t_1^{(j)} \succ t_2^{(j)} \succ \cdots \succ t_n^{(j)} \succ \phi$, where $t_1^{(j)}, t_2^{(j)}, \cdots, t_n^{(j)} \in T$ are distinct, “$\succ$” means “better than”, $\phi$ means “nothing”. Each target $t_i$, $i = 1, 2, \cdots, n$ has a strict preference over all the agents in $A$. This preference is denoted as an ordered list $P(t_i) = A_1^{(i)} \succ A_2^{(i)} \succ \cdots \succ A_m^{(i)} \succ \phi$, where $A_1^{(i)}, A_2^{(i)}, \cdots, A_m^{(i)} \in A$ are distinct.

Definition 4.1.1. An assignment $\mu$ is a set of $m$ pairs denoted as $\mu = \{(A_1, t_1'), (A_2, t_2'), \cdots, (A_m, t_m')\}$, where $t_1', t_2', \cdots, t_m' \in T$ are distinct. The notations $\mu(A_j)$ and $\mu(t'_i)$ are defined as $\mu(A_j) = t'_i$ and $\mu(t'_i) = A_j$. For any $t_i \in T - \{t_1', t_2', \cdots, t_m'\}$, define $\mu(t_i) = \phi$, which means $t_i$ is not assigned.

Remark. The assignment thus defined must exist because each subtarget is on the preference list of each agent and each agent is on the preference list of each subtarget.

Definition 4.1.2. An assignment $\mu$ is said to be **stable** if there is no blocking pair $(A_j, t_i) \not\in \mu$ for any $j = 1, 2, \cdots, m$, $i = 1, 2, \cdots, n$ such that $A_j \succ \mu(t_i)$ in $P(t_i)$ and $t_i \succ \mu(A_j)$ in $P(A_j)$.

Remark. Any stable assignment cannot be locally improved by exchanging the mates of $A_j$ and $A_i$.

Definition 4.1.3. A stable assignment $\mu^*$ is said to be **undominated** if for any $t_i$, $i = 1, 2, \cdots, n$ there is no other stable assignment $\mu$ such that $\mu(t_i) \succ \mu^*(t_i)$ in $P(t_i)$.

Remark. An undominated assignment can be viewed as one of the optimal assignments from the perspective of the subtargets. The following algorithm is to find an undominated assignment after turns of proposals and rejections.

**Algorithm 4.1.1. Proposal-Rejection algorithm for finding undominated assignment**

Step 1. For each $i = 1, 2, \cdots, n$, the subtarget $t_i$ proposes itself to the first agent in its preference list $P(t_i)$. For each $j = 1, 2, \cdots, m$, the agent $A_j$ rejects all but its most favored from its offer list if it receives more than one proposals.

Step 2. If no rejection happens, then stop; otherwise for each $i = 1, 2, \cdots, n$, the subtarget $t_i$ removes the first agent from its preference list $P(t_i)$ if the agent just rejected it.

Step 3. For each $i = 1, 2, \cdots, n$, the subtarget $t_i$ proposes itself to the first agent in its preference list $P(t_i)$ if it is just rejected. For each $j = 1, 2, \cdots, m$, the agent $A_j$ rejects all but its most favored from its offer list if it receives more than one proposals. Go to Step 2.

It can be shown that the Algorithm 4.1.1 must terminate within finite steps. In fact, at any time, no subtarget proposes to more than one agents and no subtarget proposes to one agent more than once.
This means the total number of proposals is bounded by $m \cdot n$. However, before no rejection happens, the total number of rejections is strictly monotonically increasing. And the total number of rejections is no greater than the total number of proposals, hence, there must be a step after which there is no rejection anymore, i.e. the algorithm terminates. This analysis is summarized as the following theorem.

Theorem 4.1.1. The Algorithm 4.1.1 must terminate within finite steps and the total number of rejections is no greater than $m \cdot n$.

Remark. From the practical point of view, usually, $n = O(m)$ and $m$ is a small positive integer. This implies that the total number of Proposal-Rejection operations is $O(m^2)$. The following results are from Gale and Shapley, 1962 [55, 56]. It tells that the assignment produced by the Algorithm 4.1.1 is not only stable, but also undominated.

Lemma 4.1.1. The Algorithm 4.1.1 terminates with a stable assignment.

Proof. Suppose $\mu$ is the assignment produced by the Algorithm 4.1.1. For any $j = 1, 2, \ldots, m$, $i = 1, 2, \ldots, n$ consider the pair $(A_j, t_i) \notin \mu$. If $A_j \succ \mu(t_i)$ in $P(t_i)$, then $t_i$ must have proposed to $A_j$ before being accepted by $\mu(t_i)$. Since $A_j$ finally accepts $\mu(A_j)$, hence $\mu(A_j) \succ t_i$ in $P(A_j)$, which implies that $(A_j, t_i)$ is not a blocking pair of $\mu$. Since $(A_j, t_i)$ is arbitrarily selected, $\mu$ is stable by the Definition 4.1.2.

Theorem 4.1.2. The Algorithm 4.1.1 terminates with an undominated assignment.

Proof. By the Lemma 4.1.1, we know that the output, say $\mu$, of the Algorithm 4.1.1 is a stable assignment. We claim that for any subtarget $t_i$, $i = 1, 2, \ldots, n$ it is never rejected by an agent $A_j, j \in \{1, 2, \ldots, m\}$ such that the pair $(A_j, t_i)$ belongs to some stable assignment. Consequently, $\mu$ is undominated by the Definition 4.1.3. The proof of the claim is by induction. Suppose that up to a given step in the Algorithm 4.1.1 no subtarget has ever been rejected by an agent that coexists with it in a stable assignment. Suppose at a step immediately after this point an agent $A_j$ rejects a subtarget $t_i$ in favor of another subtarget $t'_i$. Hence $t'_i \succ t_i$ in $P(A_j)$. It suffices to show that $(A_j, t_i)$ does not belong to any stable assignment. Suppose (for contradiction) that $(A_j, t_i) \in \mu'$, which is a stable assignment. Since by the inductive hypothesis, in the Algorithm 4.1.1, $t'_i$ is not rejected by any agent that coexists with it in a stable assignment, and $t'_i$ replaces $t_i$ in the offer list of $A_j$ at this step, we have that $A_j \succ \mu'(t'_i)$ in $P(t'_i)$. Hence, $(A_j, t'_i)$ is a blocking pair of $\mu'$. This contradicts the stability of $\mu'$. □

Remark. Hence at least by the Algorithm 4.1.1 an undominated assignment can be found.

4.2. Two Algorithms for Multi-Agent Exploration

This section aims to propose two algorithms for the multi-agent exploration. Both of them are based on the synchronized model (2.2.9-2.2.16). The common idea is the parallel frontier push. The difference is that in one algorithm, each agent has a common preference list over the potential subtargets; in the other algorithm, each agent has its own preference list over the potential subtargets. The communication among the agents is realized via the central map, which represents the knowledge on the world. The central map is updated when all the agents reach their assigned subtargets in each planning stage. Compared with the results on the single-agent exploration, the results on the
multi-agent exploration are more empirical and numerical. They are still in the testing phase. The more detailed theoretical analysis will be developed in the future research. In this section of this paper, we list the algorithm descriptions and some immediate analytical results. In the following sections, some results from extensive numerical experiments will be displayed. The possible work extension will be pointed out in the concluding remark.

Algorithm 4.2.1. Parallel frontier push with common preference list

Step 1. (Initialization). Choose $\varepsilon$, $0 \leq \varepsilon \leq 1$. Choose $E_k \subseteq B(s, R)$. Choose $v^{(j)}_k = s_0 \in E_k$, $j = 1, 2, \cdots, m$ as the common starting location of the $m$ agents $A_1, A_2, \cdots, A_m$. Set $O_k = \{u' | u' \in \text{Neighbor}(u), u \in E_k, u' \notin E_k, a(u') < a_0\}$, store $O_k$ as bi_heap$_k$. For each $u \in E_k$ and each $u' \in O_k$, determine $g(u, E_k)$ and $g(u', E_k)$ with $s_0$ as the starting node, let $G(u) = g(u, E_k)$ and $G(u') = g(u', E_k)$. For each $u' \in O_k$, set Pred$(u') = s_0$ (i.e. the predecessor of $u'$ is $s_0$).

Step 2. If $O_k$ is empty and $t \notin E_k$, then stop and report failure in finding target; otherwise, continue.

Step 3. $m = \min(m, |O_k|)$. Find $\{v^{(j)}_{k+1}, j = 1, 2, \cdots, m\}$ in $O_k$ as the top $m$ element in bi_heap$_k$ such that $v^{(1)}_{k+1} = \arg\min_{v \in O_k} \{\varepsilon G(v) + (1-\varepsilon)d(v, t)\}$ and $v^{(j)}_{k+1} = \arg\min_{v \in O_k \setminus \{v^{(1)}_{k+1}, \cdots, v^{(j-1)}_{k+1}\}} \{\varepsilon G(v) + (1-\varepsilon)d(v, t)\}$ for $j > 1$. For $j = 1, 2, \cdots, m$, denote $\hat{P}_{v^{(j)}_{k+1}} = v^{(j)}_k \sim \hat{a}_0 \sim \cdots \sim \hat{a}_{j-1} \sim \hat{v}, \hat{a}_i \in E_k, i = 0, \cdots, j$ as a least cost path extracted from $E_k$ from $v^{(j)}_k$ to $v$.

Step 4. Define the assignment model corresponding to the agent set $A = \{A_1, A_2, \cdots, A_m\}$ and the subtarget set $\{v^{(j)}_{k+1}, j = 1, 2, \cdots, m\}$. Let $L_j$ be the current location of $A_j$, $j = 1, 2, \cdots, m$. The preference list of $A_j$ is $P(A_j) = v^{(1)}_{k+1} \succ \cdots v^{(m)}_{k+1} \succ \phi$ for any $j \in \{1, 2, \cdots, m\}$; the preference list of $v^{(j)}_{k+1}$ is $P(v^{(j)}_{k+1}) = A^{(j)}_1 \succ \cdots \succ A^{(j)}_m \succ \phi$ for any $j \in \{1, 2, \cdots, m\}$, where $[A^{(j)}_1, \cdots, A^{(j)}_m]$ is a permutation of $[A_1, \cdots, A_m]$ such that $C(\hat{P}_{L^{(j)}_{k+1}}) \leq \cdots \leq C(\hat{P}_{L^{(m)}_{k+1}})$ with $L^{(j)}_i$ as the current location of the agent $A^{(j)}_i$, $i = 1, 2, \cdots, m$. Note that $[L^{(j)}_1, \cdots, L^{(j)}_m]$ is a permutation of $[v^{(1)}_k, \cdots, v^{(m)}_k]$.

Step 5. Solve the assignment problem in Step 4 via the Algorithm 4.1.1. For each $j = 1, 2, \cdots, m$, move the agent $A_j$ from $L_j$ to its assigned subtarget $v^{(j)}_{k+1}$ through the path $\hat{P}_{L_j, v^{(j)}_{k+1}}$. $E_{k+1} = E_k \cup \{v^{(j)}_{k+1}, j = 1, 2, \cdots, m\}$. If $t \in E_{k+1}$, then stop and report success in finding target, determine $g(t, E_{k+1})$ by the A* algorithm; otherwise, continue.
Step 6. $bi_{heap_{k+1}} = bi_{heap_k}$. For each $j = 1, 2, \cdots, m$, for each $v \in \text{Neighbor}(v^{(j)}_{k+1}), v \not\in E_{k+1}$ evaluate $a(v)$. If $a(v) \geq a_\infty$, ignore $v$; if $a(v) < a_\infty$ and $v \not\in O_k$, then $O_{k+1} = (O_k \setminus \{v^{(j)}_{k+1}\}) \cup \{v\}$, update $bi_{heap_{k+1}}$ by removing $v^{(j)}_{k+1}$ and entering $v$, let $G(v) = G(v^{(j)}_{k+1}) + C(v^{(j)}_{k+1}, v)$, and set $\text{Pred}(v) = v^{(j)}_{k+1}$; if $a(v) < a_\infty$ and $v \in O_k$, then $O_{k+1} = O_k \setminus \{v^{(j)}_{k+1}\}$, let $G(v) = G(v^{(j)}_{k+1}) + C(v^{(j)}_{k+1}, v)$ and set $\text{Pred}(v) = v^{(j)}_{k+1}$ if $G(v) > G(v^{(j)}_{k+1}) + C(v^{(j)}_{k+1}, v)$, update $bi_{heap_{k+1}}$ by repairing the current $bi_{heap_{k+1}}$. $k = k + 1$. Go to step 2.

Compared with the Algorithm 3.2.1, other than expanding the explored region by one node in each planning stage, the Algorithm 4.2.1 expands the explored region by multiple nodes in each planning stage. Since in the $k$th planning stage, the node in $O_k$ with the smallest $f$ label is assigned to some agent, and the $G$ label for any node is non-increasing if it exists, we have the immediate result as same as the Lemma 3.1.1 and the Theorem 3.2.1. That is:

Theorem 4.2.1. In the Algorithm 4.2.1, if for some $k \in \mathbb{Z}^+, k > 0$, $O_k$ is empty and $t \notin E_k$, then there is no path from the base to the target $t$. If there exists a path from the base to the target $t$, then there must be a $N \in \mathbb{Z}^+, N > 0$, such that $t \in E_N$.

An obvious advantage of the expansion by multiple nodes in frontier is that the knowledge on the world grows faster when compared with the single-agent exploration. Also, when multiple agents are involved in the exploration, the exploration “burden” is allocated to every agent. The result may be the improved swiftness on the target finding. No analytical result for comparing the efficiency of a single agent with that of multiple agents is available now since the performance is highly case-dependent. Some simulation results will be presented in the later sections. There is at least one disadvantage of applying the Algorithm 4.2.1 to the multi-agent exploration. Since the selected subtargets in each planning stage may be very close to each other, the risk that all the agents enter one dead-end corner cannot be ignored. The following algorithm provides a remedy.

Algorithm 4.2.2. Parallel frontier push with individual preference list

Step 1. (Initialization). Choose $\varepsilon$, $0 \leq \varepsilon \leq 1$. $k = 0$. Choose $v^{(j)}_k = s^{(j)}_0 \in B(s, R), j = 1, 2, \cdots, m$ as the starting locations of the $m$ agents $A_1, A_2, \cdots, A_m$ such that $s^{(1)}_0, s^{(2)}_0, \cdots, s^{(m)}_0$ are distinct. Set $E_k = \{s^{(1)}_0, s^{(2)}_0, \cdots, s^{(m)}_0\}$. Set $O^{(j)}_k = \{u \mid u \in \text{Neighbor}(s^{(j)}_0), a(u) < a_\infty\}$, store $O^{(j)}_k$ as
Step 5. Solve the assignment problem in Step 4 via the Algorithm 4.1.1. Suppose the solution matches
the predecessor of \( u \) is \( s_0^{(j)} \) if \( G(u) > C(s_0^{(j)}, u) \).

Step 2. For each \( j = 1, 2, \cdots, m \), if \( O_k^{(j)} \) is empty and \( t \notin E_k \), then stop the agent \( A_j \), \( m = m-1 \). If each

\[ bi\_heap_k^{(i)} \]. For each \( u \in \bigcup_{j=1}^m O_k^{(j)} \) set \( G(u) = M \), where \( M \) is a sufficiently large number. For

each \( j = 1, 2, \cdots, m \), for each \( u \in O_k^{(j)} \), let \( G(u) = C(s_0^{(j)}, u) \) and set \( Pred(u) = s_0^{(j)} \) (i.e. the
Step 3. For each \( j = 1, 2, \cdots, m \), find \( u_0^{(j)} = \arg \min_{v \in O_k^{(j)}} f(v) = \varepsilon G(v) + (1-\varepsilon)d(v, t) \) as the top element in

\[ bi\_heap_k^{(i)} \]. \( C_k = \bigcup_{j=1}^m \{ u^{(j)} \} \). If \( |C_k| < m \), for each \( j = 1, 2, \cdots, m \), find \( u_0^{(j)} = \arg \min_{v \in O_k^{(j)}, v \neq u^{(j)}} f(v) = \varepsilon G(v) + (1-\varepsilon)d(v, t) \) as the next top element in \( bi\_heap_k^{(i)} \). \( C_k = C_k \bigcup \bigcup_{j=1}^m \{ u^{(j)} \} \). Continue this
until \( |C_k| \geq m \). For \( j = 1, 2, \cdots, m \), denote \( \hat{P}_{i-1}^{(j)} = v_k^{(j)} \sim \hat{\alpha}_1 \sim \cdots \sim \hat{\alpha}_j \sim v \), \( \hat{\alpha}_i \in E_k, i = 0, \cdots, r_k^{(j)} \) as a least cost path extracted from \( E_k \) from \( v_k^{(j)} \) to \( v \).

Step 4. Define the assignment model corresponding to the agent set \( A = \{ A_1, A_2, \cdots, A_m \} \) and
the subtarget set \( C_k \). Let \( L_j = v_k^{(j)} \) be the current location of \( A_j, j = 1, 2, \cdots, m \). The preference list of
\( A_j \) is \( P(A_j) = u^{(j)} \succ u^{(j)} \succ \cdots \succ \phi \) for any \( j \in \{ 1, 2, \cdots, m \} \); the preference list of any \( u \in C_k \) is
\( P(u) = A_{1}^u \succ \cdots \succ A_{m}^u \succ \phi \), where \( [ A_{1}^u, \cdots, A_{m}^u ] \) is a permutation of \( [ A_1, \cdots, A_m ] \) such that
\( C(\hat{P}_{k-1}^{(i), u}) \leq \cdots \leq C(\hat{P}_{k}^{(i), u}, u) \) with \( L_i^u \) as the current location of the agent \( A_i^u \), \( i = 1, 2, \cdots, m \).

Note that \( [ L_1^u, \cdots, L_m^u ] \) is a permutation of \( [ v_1^{(1)}, \cdots, v_m^{(m)} ] \).

Step 5. Solve the assignment problem in Step 4 via the Algorithm 4.1.1. Suppose the solution matches
\( A_j \) with \( v_k^{(j)} \), \( j = 1, 2, \cdots, m \). For each \( j = 1, 2, \cdots, m \), move the agent \( A_j \) from \( v_k^{(j)} \) to \( v_k^{(j)} \)
through the path \( \hat{P}_{k-1}^{(i), v_k^{(j)}}, E_{k+1} = E_k \cup \{ v_k^{(j)}, j = 1, 2, \cdots, m \} \). If \( t \in E_{k+1} \), then stop and report
success in finding target, determine \( g(t, E_{k+1}) \) by the A* algorithm; otherwise, continue.
Step 6. For \( j = 1, 2, \cdots, m \), \( bi_{\text{heap}}_{k+1}^{(j)} = bi_{\text{heap}}^{(j)} \). For each \( j = 1, 2, \cdots, m \), for each \( v \in \text{Neighbor}(v^{(j)}_{k+1}) \), \( v \notin E_{k+1} \) evaluate \( a(v) \). If \( a(v) \geq a_{\infty} \), ignore \( v \); if \( a(v) < a_{\infty} \) and \( G(v) \) does not exist, then \( O_{k+1}^{(j)} = (O_{k}^{(j)} - \{ v^{(j)}_{k+1} \}) \cup \{ v \} \), update \( bi_{\text{heap}}_{k+1}^{(j)} \) by removing \( v^{(j)}_{k+1} \) and entering \( v \), let \( G(v) = G(v^{(j)}_{k+1}) + C(v^{(j)}_{k+1}, v) \), and set \( \text{Pred}(v) = v^{(j)}_{k+1} \); if \( a(v) < a_{\infty} \) and \( G(v) \) does exist, then \( O_{k+1}^{(j)} = (O_{k}^{(j)} - \{ v^{(j)}_{k+1} \}) \cup \{ v \} \), let \( G(v) = G(v^{(j)}_{k+1}) + C(v^{(j)}_{k+1}, v) \) and set \( \text{Pred}(v) = v^{(j)}_{k+1} \) if \( G(v) > G(v^{(j)}_{k+1}) + C(v^{(j)}_{k+1}, v) \), update \( bi_{\text{heap}}_{k+1}^{(j)} \) by repairing the current \( bi_{\text{heap}}_{k+1}^{(j)} \). \( k = k + 1 \). Go to step 2.

Lemma 4.2.1. In Step 5 of the Algorithm 4.2.2, the first subtarget in each preference list is assigned.

Proof. Since the first subtarget in each preference list keeps proposing to an agent if it is rejected and its preference list is still not empty, it must be able to stay in the offer list of some agent without being rejected, i.e. it is finally assigned. \( \square \)

Remark. By this lemma, together with the fact that the \( G \) label for any node is non-increasing if it exists, we have the immediate result as same as the Lemma 3.1.1 and the Theorem 3.2.1. That is:

Theorem 4.2.2. In the Algorithm 4.2.2, if for some \( k \in \mathbb{Z}_{>0} \), \( O_{k}^{(j)} \) is empty for each \( j = 1, 2, \cdots, m \) and \( t \notin E_{k} \), then there is no path from the base to the target \( t \). If there exists a path from the base to the target \( t \), then there must be a \( N \in \mathbb{Z}_{>0} \), such that \( t \in E_{N} \).

Remark. Compared with Algorithm 4.2.1. The Algorithm 4.2.2 is more “distributed”. There does exists some example in which it performs much better than the Algorithm 4.2.1.

5. Numerical Experiments for Single-Agent Exploration

In this section, two typical terrain models are employed to test the Algorithm 3.2.1. One is the uniformly generated random terrain; the other is the fractally generated random terrain. As mentioned in the section 2.2, the eight-connected square grid network is used as the discretization of the world. In the uniformly generated random terrain, the probability that the reachability of each node exceeds a predetermined threshold is fixed. (e.g. 0.35 in the following examples). And some disk-shape forbidden regions with the radius between two to five times of the detection range of the agent are uniformly generated and embedded in. In the fractally generated random terrain, the diamond square algorithm (see Miller and Gavin, 1986 [57]) is employed. The sizes of the terrain matrices are 300×300 and 257×257 for the uniformly generated random terrain and the fractally generated random terrain respectively. The simulations of terrain exploration and the path finding are coded with Matlab 7.1 and performed in Dell PC with P4 3.20 G CPU and 1.0G DDR2 MEM. The tests include comparing the total traveling costs of the agent upon finding the target under the different values of \( \varepsilon \).
Figure 5.1. Single-agent exploration and path finding in a uniformly generated random terrain under the commands of the Algorithm 3.2.1

Figure 5.2. Performance Comparison of Single-agent exploration and path finding in a uniformly generated random terrain (as in the Figure 5.1) under the commands of the Algorithm 3.2.1
Figure 5.3. Single-agent exploration and path finding in a fractally generated random terrain under the commands of the Algorithm 3.2.1

Figure 5.4. Performance Comparison of Single-agent exploration and path finding in a fractally generated random terrain (as in the Figure 5.3) under the commands of the Algorithm 3.2.1
and comparing the qualities of the respective paths finally found. In the tests, the agent’s total traveling cost is measured with relative value, i.e. the ratio of the agent’s total traveling cost at some $\varepsilon$, $0 \leq \varepsilon \leq 1$ to the agent’s total traveling cost at $\hat{\varepsilon} = \frac{1}{2}$. The length of the path found by the agent is also measured with relative value, i.e. the ratio of the length of path found at some $\varepsilon$, $0 \leq \varepsilon \leq 1$ to the length of path found at $\hat{\varepsilon} = \frac{1}{2}$. There are two final paths found by the agent. One is the predecessor path, which is reconstructed via backtracking; the other is the extracted path, which is computed according to the final map of the world. Since the second path is computed by the A* algorithm, its length is conditionally minimum. The predecessor path is no “shorter” than the extracted path. When $\varepsilon = \hat{\varepsilon} = \frac{1}{2}$, both paths reach the absolutely “shortest”. Some results from a uniformly generated

![Figure 5.5 (a). Average relative length of path in uniformly generated random terrains](image)

![Figure 5.5 (b). Logarithm of the average relative traveling cost in uniformly generated random terrains](image)

![Figure 5.5 (c). Average relative length of path in fractally generated random terrains](image)

![Figure 5.5 (d). Logarithm of the average relative traveling cost in fractally generated random terrains](image)

Figure 5.5. Some statistics on multiple simulations of single-agent exploration and path finding in uniformly generated random terrains and fractally generated random terrains under the commands of the Algorithm 3.2.1
random terrain and a fractally generated random terrain are presented as the Figures 5.1 through 5.4. The Figures 5.1 and 5.3 show four paths corresponding to \( \varepsilon = 0, 0.167, 0.333, 0.5 \) respectively in the two terrains. The black color stands for the forbidden region, the gray color stands for the unexplored region, and the white color stands for the final explored regions, i.e. the final maps. The central solid curves are the conditionally shortest paths extracted from the corresponding final maps. The Figures 5.2 (a) and the 5.4(a) show the path comparisons in the two terrains; the Figures 5.2 (b) and 5.4 (b) show the comparisons of the total traveling costs, in which the common logarithm (with base 10) is used. From the practical point of view, these comparison results support selecting the value of \( \varepsilon \) close to zero, i.e. close to the depth-first exploration.

Also, more than 100 simulations are performed for each of the two types of terrain with different values of \( \varepsilon \). The statistics on the final extracted conditionally shortest paths and the total traveling costs are shown in the Figure 5.5. The length of the error bar is twice of the standard deviation. The statistics shows that \( \varepsilon = 0 \) (depth-first exploration) has very competitive efficiency while the corresponding final path found is not too “longer” than the absolute shortest one. In the next section we will apply the depth-first exploration to an exploration team.

6. Numerical Experiments for Multi-Agent Exploration

In this section, we apply the depth-first exploration (\( \varepsilon = 0 \) in both the Algorithms 4.2.1 and the Algorithm 4.2.2) to an exploration team with up to six agents. The context of the terrain generation is as same as that used for single agent. The simulations are also performed in Dell PC with P4 3.20 G CPU and 1.0G DDR2 MEM. The tests include displaying and comparing the efficiency and the effectiveness of the Algorithms 4.2.1 and the Algorithm 4.2.2 at \( \varepsilon = 0 \). The efficiency is measured with the leading traveling cost, which is the total traveling cost of the agent that finds the target. The effectiveness is measured with the length of the final conditionally shortest path extracted from the final map. The performance of the team with different number of agents is also compared. The Figures 6.1 through 6.8 are testing examples of the Algorithms 4.2.1 and the Algorithm 4.2.2 for multi-agent exploration and path finding in the uniformly generated random terrains and the fractally generated random terrains. The overall performance of the Algorithm 4.2.2 seems to be better than that of the Algorithm 4.2.1 except for the terrain example shown in the Figures 6.1 and 6.2. In that example, the team of agents under the commands of the Algorithm 4.2.1 are congregated together and happen to find a very good “corridor”. The same team under the commands of the Algorithm 4.2.2 are dispersed with some agents exploring away from the best corridor. However, the dispersion has the advantage of hedging the risk that all the agents are stuck in one bad corner. A representative terrain example is shown in the Figures 6.3 and 6.4. The Figure 6.3 (a) shows that the agent team, with the number of the members up to six, under the commands of the Algorithm 4.2.1 always explore the upper triangular part of the terrain. Consequently, the leading traveling cost is much higher and the final path found is much longer compared with what the same agent team under the commands of the Algorithm 4.2.2 can achieve. Note that the Figure 6.4 (b) shows that the leading traveling cost of the team drops significantly when the number of the members of the team rises from one to two. The reason is that one agent enters a good corridor in the lower triangular part of the terrain at the sacrifice of the other
Figure 6.1. Multi-agent exploration and path finding in a uniformly generated random terrain under the commands of the Algorithm 4.2.1 at $\varepsilon = 0$.

Figure 6.2. Multi-agent exploration and path finding in a uniformly generated random terrain under the commands of the Algorithm 4.2.2 at $\varepsilon = 0$. 
Figure 6.3 (a). Exploration by 6 agents

Figure 6.3 (b). Performance comparison

Figure 6.3. Multi-agent exploration and path finding in a uniformly generated random terrain under the commands of the Algorithm 4.2.1 at $\epsilon = 0$.

Figure 6.4 (a). Exploration by 6 agents

Figure 6.4 (b). Performance comparison

Figure 6.4. Multi-agent exploration and path finding in a uniformly generated random terrain under the commands of the Algorithm 4.2.2 at $\epsilon = 0$. 
Figure 6.5. Multi-agent exploration and path finding in a fractally generated random terrain under the commands of the Algorithm 4.2.1 at $\epsilon = 0$.

Figure 6.6. Multi-agent exploration and path finding in a fractally generated random terrain under the commands of the Algorithm 4.2.2 at $\epsilon = 0$. 
Figure 6.7. Multi-agent exploration and path finding in a fractally generated random terrain under the commands of the Algorithm 4.2.1 at $\epsilon = 0$.

Figure 6.8. Multi-agent exploration and path finding in a fractally generated random terrain under the commands of the Algorithm 4.2.2 at $\epsilon = 0$. 
agent that keeps exploring the upper triangular part of the terrain.

Although the testing examples indicate the more reliable performance of the Algorithm 4.2.2. The conclusion that the Algorithm 4.2.2 is better cannot be drawn. One reason is that even if an agent can avoid exploring some bad areas at the sacrifice of some other agents it still has the chance of being stuck in some dead corner as it moves. Without a companion in its vicinity, it may still take long time to escape from the dead-end.

7. Summery, Conclusions, and Discussions
This paper addresses the problem of path finding in unknown environment based on exploration agents. Without any prior information, it’s impossible to predetermine a reference path. Different from the idea of replanning algorithms, this paper explores the idea of the map expansion through the agents’ exploration. The central idea is to start from a very small map within the base area (i.e. the area from which the agents start) and push the boundary of the map (i.e. the frontier of the explored regions) by selecting nodes from the candidate list for the agents to move to. The exploration ends upon the moment the target appears in the map. Based on this idea, dynamic bi-objective models are proposed for the single-agent based exploration and the multi-agent based exploration. The models can be viewed as multi-stage decision planning problems. For the multi-agent based exploration, the model is synchronized, and in each planning stage an assignment model is proposed to resolve the goal conflict. Since the time complexity of solving the assignment submodel is bounded by the product of the number of the agents and the number of the subtargets in each planning stage, the idle time of each agent due to the subtarget allocation can be neglected. The idle time of each agent consists mainly of the time during which the agent waits for other agents, if there is, to reach their respective assigned subtargets. Although in synchronized multi-agent exploration model the idle time in is inevitable, it’s easy to update the cumulative map and pass the information to each agent in accordance with the commands of the associated algorithm.

This paper first extensively studies the single-agent exploration. In each planning stage, the agent faces multiple alternatives for moving on. Intuitively, in order to select a frontier node, three questions should be raised, they are: how close this node is to the target? how far away it is from the starting node? and how much the cumulative traveling cost will be if the agent moves to that node? In this paper, a score as the weighted average of these three factors is used to rank the priority of the nodes in the candidate list. Studies show that the three factors are not independent. To guarantee the completeness of the exploration, the third factor is dropped from the evaluation function, however, studies also show that changing the weights of the first two factors significantly influences the value of the third factor. Extensive simulations with the uniformly generated random terrains and the fractally generated random terrains show that the performance at $\epsilon = 0$, or equivalently the depth-first exploration, is competitively good with the guaranteed completeness of exploration, the relative small exploration efforts, and, in most cases, the acceptable final path quality. It should be noted that although at $\epsilon = \frac{1}{2}$ it is guaranteed that an absolutely optimal path can be found with the minimum
number of nodes reached, the traveling cost of the agent is still considerably high. Simulations with the uniformly generated random terrains and the fractally generated random terrains show that to reduce the value of $\varepsilon$ is a way to reduce the traveling cost. However, this saving is at the sacrifice of the optimality of the final path found. This experimental observation agrees with the experimental performance of the weighted heuristic used by Likhachev, 2005 [58] to develop the AA* (any time A*) algorithm to search a suboptimal path in an extremely large directed graph with positive edge weights. However, open questions like the Conjecture 3.2.1 and why when $\varepsilon < \frac{1}{2}$ the total traveling cost keeps decreasing (even drastically) as $\varepsilon$ is reduced further are not completely answered.

The algorithms for the multi-agent exploration are naturally extended from the algorithm for the single-agent exploration. One version of extension is the parallel frontier push with a common preference list; the other version of extension is the parallel frontier push with individual preference lists. The multi-agent simulation results with the two types of terrains show that at $\varepsilon = 0$ the efficiency of finding the target can be significantly improved when more than one agent is used. This experimental observation agrees with the common sense that multiple agents can be more efficient than a single agent if the multiple agents are controlled by a proper coordination mechanism. The author of this paper suggests an analytical study that verifies this within a probabilistic theoretical framework. Compared with the method for the single-agent based exploration, the methods for the multi-agent based exploration are still in the testing phase. The only analytical results are the provable completeness of the exploration. Based on the simulation results, the Algorithm 4.2.2 is more “distributed” and seems to outperform the Algorithm 4.2.1. However, a like conclusion cannot be drawn. The author of this paper suggests a study on more algorithms for multi-agent exploration.

This paper also assumes that once the agents stop the exploration with a path from the base to the target found, the path can be used for further navigation. However, from the practical point of view, this is not always true. Consider in a dynamic environment, such path could be destroyed such that replanning must be considered. The author of this paper suggests using the D* algorithm [40] or D* Lite algorithm [41] for further navigation. The initialization in both algorithms could be the final map constructed via the multiple agents’ exploration.

**Acknowledgement**

The author of this paper thanks Professor Shih-Ping Han and Professor Daniel Q. Naiman for their helps. This work is supported by the Donald Fink Fund for Scientific Computation and the Charles Counselman Fellowship for Ph.D research work in department of Applied Mathematics and Statistics, the Johns Hopkins University.
References


Amsterdam.


