# Collaborative Search for Multi-goal Pathfinding in Ubiquitous Environments

Oudom Kem, Flavien Balbo, and Antoine Zimmermann

Univ Lyon, MINES Saint-Étienne, CNRS, Laboratoire Hubert Curien, UMR 5516, F-42023, Saint-Étienne, France
{oudom.kem,flavien.balbo,antoine.zimmermann}@emse.fr

**Abstract.** Multi-goal pathfinding (MGPF) is a problem of searching for a path between an origin and a destination, which allows a set of goals to be satisfied. We are interested in MGPF in ubiquitous environments that are composed of cyber, physical and social (CPS) entities from connected objects, to sensors and to people. Our approach aims at exploiting data from various resources such as CPS entities and the Web to solve MGPF. However, accessing resources creates overheads – specifically latency affecting the efficiency of the approach. In this paper, we present a collaborative multi-agent search model that addresses the latency problem. The model handles the process of accessing resources such that agents are not blocked while data from resources are being processed and transferred. Agents search concurrently and collaboratively on different parts of the search space. The model exploits the knowledge and structure of the search space to distribute the work among agents and to create an agent network facilitating agent communications as well as separating the search from the communications. To evaluate our model, we apply it in uniform cost search, creating a collaborative uniform cost algorithm. We compare it to the original algorithm. Experiments are conducted on search spaces of various sizes and structures. In most cases, collaborative uniform cost is shown to run significantly faster and scale better in function of latency as well as graph size.

**Keywords:** Collaborative search, Multi-Agent search, Multi-goal pathfinding, Ubiquitous environments

## 1 Introduction

Pathfinding is a problem that has been studied extensively due to its importance in various fields such as AI, robotics, logistics and video games. There are different variations of pathfinding problems [2, 8] such as single-agent pathfinding, multi-agent pathfinding in static, dynamic and real-time environments. Numerous techniques have been proposed to address pathfinding [1]. In this paper, we focus on a particular kind of pathfinding problem, multi-goal pathfinding (MGPF) in the context of ubiquitous environments accommodating cyber, physical and social (CPS) entities such as sensors, smart objects and humans.

MGPF is a problem of searching for a path between an origin and a destination, which allows a set of goals to be satisfied. Our approach to solving MGPF exploits data acquired from CPS entities in a given environment and from external resources such as the Web. It uses up-to-date and dynamic information from various resources for path computation. For path evaluation, we use generic criteria, which are not limited to distance, and quality of entities, which is determined using qualitative information from resources. To understand the underlying motivation of our approach, consider the following scenario. A traveler, Carol, arrives at an airport. Carol wants to find a path to her departure gate. Carol has a set of activities (goals) she wants to do on her way to the gate: get a trolley for her luggage, check-in, buy a takeout for lunch and find a waiting seat near a power socket to charge her laptop. Using spatial information of the airport, we can find a path to the gate. Information about the airport makes it possible to determine which locations allow Carol to satisfy each of her goals. For example, restaurant is a business which prepares and serves food and drinks to customers in exchange for money. By obtaining that piece of information from the Web, we are able to deduce that Carol can buy lunch at locations of type restaurant. Dynamic and up-to-date information from sensors and smart objects enables us to determine the optimal path for Carol. For instance, instead of going to a trolley area, which is at the opposite direction of her gate, it is possible to locate an available trolley nearby that was left by other people, thanks to data from connected trolleys. We might suggest Carol to take an escalator instead of an elevator because we know that there are too many people in the queue waiting for the elevators or that the elevators are out of service thanks to the feeds from sensors. In addition, information from social entities such as other travelers or personnel can be used to enhance Carol's travel experience. For instance, reviews by travelers (e.g. quality or availability) on restaurants enable us to choose locations that are at Carol's best interests.

Considering the aim of the approach, one might ask two challenging questions: (1) Which resources to use to solve a MGPF problem? (2) How to deal with the dynamics, mobility and heterogeneity of CPS entities? The first question is concerned with the discovery of resources that are relevant to a given MGPF problem. We address this question via the use of a data model to capture necessary knowledge enabling resource discovery. Regarding the second question, there are existing works that address these issues in the context of IoT. As an example, in [4], the authors propose a multi-agent-based socio-technical network (STN) to manage the complexity of CPS entities. In our approach, we focus on the conceptual level, and we employ one of the existing solutions such as STN to abstract away the complexity at the lower level.

In our approach, we solve MGPF by abstracting a given environment as a search space and use search algorithms to find the path. Necessary information for finding and evaluating a path during a search is acquired from various resources from CPS entities to the Web. Accessing resources creates overheads resulted from the latency of processing and transferring data from their sources. To address this issue, we propose a collaborative search model for search algo-

rithms, which is the main contribution of this paper. The model is composed of multiple agents collaboratively and concurrently searching on different parts of a search space. It handles the process of resource accesses such that agents do not have to wait for data and are able to perform other tasks while requests to resources are being processed. It exploits the structure and knowledge of the search space to distribute the work efficiently among agents and to construct an agent network on-the-fly to facilitate agent communications and separate the search from the communications.

The rest of the paper is organized as follows. First, we review related work. Second, we present the problem addressed in this paper. Third, a detailed description of the collaborative search model is provided. Fourth, we present an application of our model in uniform cost search, and provide some experimental results. Fifth, we conclude the paper and outline future work.

## 2 Related work

There are two common variants of MGPF. First, given a single start and multiple goals, MGPF is defined as a problem of searching for paths for each start-goal pair, resulting in multiple paths [12]. Second, MGPF is treated as a traveling salesman problem (TSP) in which the aim is to find a path from a start to a number of goals before reaching the destination such as in [5, 13]. Our problem is close to the second definition. However, unlike the classical TSP, we have constraints on the order of goals to satisfy. The work in [21] addresses a TSP with partial order constraints. The author proposes two algorithms to solve the selection and ordering of points-of-interests (goals), which are places, for indoor navigation systems. Path computation is based on complete spatial knowledge of an environment, and distance is the sole criterion for path evaluation. In this paper, we address a problem similar to [21], but the specific property of our problem is that satisfying a goal is not limited to passing by a place, but can be any activity carried out via a CPS entity, which can be mobile and dynamic. Furthermore, we have a strict order in which goals are satisfied.

There is a rich body of literature on pathfinding. Many search algorithms have been proposed to address various aspects of the problem. The most common search algorithms are the centralized and synchronous ones such breadth-first search, depth-first search, Dijkstra's algorithm, uniform cost search and A*. A* is probably the most used heuristic algorithm due to its theoretical properties that guarantee completeness and optimality [9], provided that the heuristics are consistent. Considerable efforts have been invested to optimize A* by reducing search space [3], mitigating memory requirements [15] and adapting it to dynamic environments [11, 18]. In these algorithms, each step is performed sequentially as the global state of the search is required. For example, A* selects a promising node to expand by comparing it to all the candidate nodes. With latency, sequential search will become impractical as the algorithm is blocked while waiting for requested data to determine the cost of each node.

Much work has been done [16, 17, 20] to address search that involves multiple agents, commonly known as cooperative pathfinding or multi-agent pathfinding. However, such work aims at finding non-interfering paths for each agent from their current state to their respective goal state. In our case, each agent may have different starts, but they cooperate to find the same goal state. This falls under the definition of collaborative mulit-agent search as classified by the author in [6]. Parallelization techniques such as [10] and [19] have been used to improve search. They distribute workloads or search operators between processors using generic mechanisms independent of the problem (e.g. using hash function to assign each node to a process). Furthermore, in [14], the authors propose a multi-agent A* based on agents possessing different search operators. Workloads are distributed based on the operators (i.e. discovered nodes are sent to agents who have the operators to expand them). In our model, we separate a search space among agents based on the structure of the space, thus distributing the workload by assigning each agent a sub search space. In addition, to the best of our knowledge, there is no work that addresses the latency during search, which makes reasonable sense since accessing resources to compute node cost is specific to our approach.

## 3 The problem

In this section, we present our method to abstract a ubiquitous environment into a search space. Then, we provide an overview of our approach, and describe how search algorithms are positioned in the approach.

### 3.1 Environment abstraction

Spatial information is necessary but insufficient to determine the locations at which a goal can be satisfied. Up-to-date and qualitative information from CPS entities and external resources are also required to find an optimal path. Therefore, in our approach, we model a ubiquitous environment by integrating its spatial and CPS dimensions along with the notion of resources. We assume that the spatial topology of a ubiquitous environment is abstracted as a graph $SG$ where nodes are locations in the environment. A directed edge between two nodes $n$ and $n'$ is defined if, in the given environment, the location represented by $n'$ is directly accessible from that by $n$.

**Definition 1.** ***Environment***
*An* environment *at a given time is a tuple* $E_t = (SG, HE, CPSE, R)$ *where:*

- $SG$ *is a search graph defined as* $SG = \langle L, C \rangle$ *where* $L$ *is a finite set of nodes representing locations in* $E$ *and* $C \subseteq L \times L$ *is a set of edges representing connections between locations*
- $HE$ *represents an organizational hierarchy of* $E$*. It is a tree whose elements correspond to hierarchy entities (e.g. terminals) of* $E$ *and the child relation indicates sub-hierarchy entities (e.g. zones within a terminal). Locations are*

*grouped under hierarchy entities, so the leafs of HE are directly connected to the locations*
- *CPSE is a finite set of CPS entities located in E*
- $R = (r_n)_{n \in CPSE \cup C}$ *is a finite set of resources providing information about a CPS entity or giving information on how to move between locations. An example of a resource can be a website, a database or an API to sources of data collected from cyber-physical entities.*

Fig. 1 shows an example of an abstracted environment. The top layer is the spatial dimension of an environment. The bottom layer consists of resources relevant to an environment. The middle layer integrates $L$ with $CPSE$, and connects them to $R$.

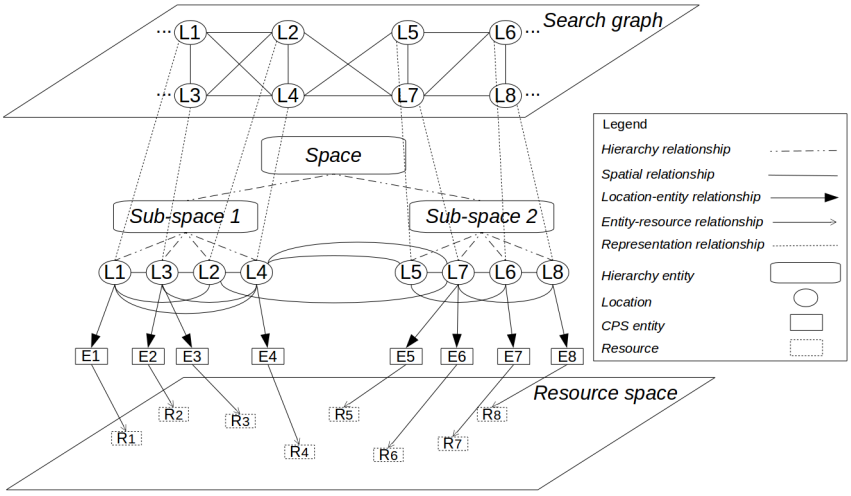


**Fig. 1.** An example of an environment description

**Definition 2.** ***Multi-goal pathfinding***
*By abstracting a given environment as previously described, we can formulate MGPF as a tuple* $MGPF = (E_t, n_o, n_d, G, CR, f)$ *where* $E_t$ *is a representation of an environment at time* $t$, $n_o \in L$ *is a node representing the start location,* $n_d \in L$ *is a node representing the destination,* $G$ *is an ordered list of goals to satisfy,* $CR$ *is a set of criteria for evaluating a path and* $f$ *is a cost function used to evaluate paths. Criteria are problem-specific. For instance, a criterion can be distance, price, duration or all of them combined.* $f$ *determines how* $CR$ *is taken into account in the decision process when choosing a path (e.g. prioritize a subset of* $CR$ *or compromise all the criteria in* $CR$*). A problem is solved when an optimal path is found. A path is a list of locations through which every goal can be satisfied in the given order. A path is optimal if it has a minimum cost evaluated using* $f$*.*

### 3.2 The approach

Our approach to MGPF consists of two main steps *goal-space graph generation* and *multi-layer search*. In the first step, we construct a goal-space graph to represent goals, the locations where each goal can be satisfied and the order of goals to satisfy. A goal-space graph, denoted by $\pi$, is an acyclic graph where nodes are goal-location pairs, and nodes are connected according to the order of goals defined in $G$. In this work, we associate a goal $g$ to an activity $a^g$. We say that $g$ can be satisfied at a location $l$ if $l$ contains at least one entity $cpse \in CPSE$ through which $a^g$ can be carried out. For illustration purpose, suppose the followings are the locations where Carol's goals can be satisfied: trolley $= \{l_{21}, l_{32}\}$, check-in $= \{l_{43}, l_{64}\}$, lunch $= \{l_{15}, l_{61}, l_{11}\}$ and waiting seat $= \{l_{20}, l_{71}\}$. We can generate a goal-space graph $\pi$ as shown in Fig. 2.

In the second step, we search over $\pi$ to find an optimal path. $\pi$ is an abstract graph built on top of $SG$. An edge of $\pi$ is equivalent to a path that may consist of multiple nodes on $SG$. For instance, an edge between $(lunch, l_{15})$ and $(waitingseat, l_{20})$ may be equivalent to a sequence of nodes $(l_{15}, l_{16}, l_{28}, l_{18}, l_{20})$ on $SG$. Computing the cost of an edge between two nodes of $\pi$ is equivalent to a pathfinding problem of two corresponding nodes on $SG$. Searching on $SG$ requires accesses to resources to determine the cost of moving between nodes, which leads to the issue of latency. In this paper, we address the search on $SG$ by providing a collaborative search model that can be used to adapt search algorithms to efficiently handle latency and to improve search efficiency.

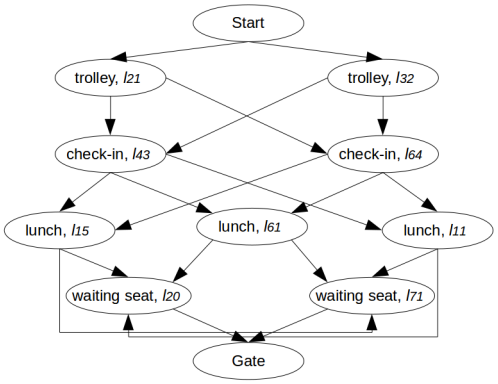


**Fig. 2.** An example of a goal-space graph

## 4 Collaborative search model

The aim of the collaborative search model is to manage resource accesses, thus the consequential latency, and to improve search efficiency. This model can be

applied to forward-search algorithms such as breadth-first search, depth-first search, uniform cost search and A*. Generally, during each iteration, a forward-search algorithm *selects* a node from the list of candidate nodes $Frontier$, *generates* its child nodes, *prunes* some unpromising nodes, and *updates* $Frontier$ to include the remaining children [7]. The critical point that creates latency is the *generation of child nodes*. To determine the cost for moving from an expanded node $n$ to its child node $n'$, we need to retrieve information from one or multiple resources. An example of the cost can be time, distance or all of them combined. Besides enabling us to determine the value of the cost, dynamic information from resources allows us to factor in cost-influencing criteria such as the fact that an elevator in the path is out of service or the path is currently blocked. This process results in latency as the algorithm has to wait for the requested information to arrive to determine the cost of child nodes and proceed the execution. The proposed model adapts each step of forward-search algorithms. More precisely, to distribute workloads, agents explore different parts of a search space concurrently by executing a search algorithm, thus *selecting* nodes from their respective part of the search space. *Child generation* is modified into an asynchronous and non-blocking process where agents are able to execute other tasks while information from resources is being retrieved. Discovered paths to a node in another part of the search space are communicated to agents responsible for that part so that they can *prune* and *update* their local search process.

**Definition 3.** ***Collaborative search model***
*A collaborative search model is a tuple* $CSM = (E_t, n_o, n_d, SA, Sa^o, RA, NA)$ *where* $E_t$ *is an abstraction of an environment at time* $t$ *as previously defined,* $n_o \in L$ *is a node representing a start location,* $n_d \in L$ *is a node representing a destination location,* $SA$ *is a set of search agents executing the search algorithm,* $Sa^o \in SA$ *is the initial search agent that starts the search process,* $RA$ *is a set of resource agents, each of which is responsible for retrieving information from a set of resources, and* $NA$ *is a set of network agents. A network agent is responsible for managing a search process and related communications within the coverage of a set of hierarchy entities.*

The model is based on collaborations among search agents, resource agents and network agents. We describe in the following subsections the roles of these agents in realizing the aim of this search model.

### 4.1 The search agents

In our model, the role of search agents is to execute a given search algorithm. The execution cycle of a search agent consists of *expanding a node*, *retrieving the cost of a child node* and *processing received messages*. A search agent $Sa$ is responsible for exploring a part of the search space. The nodes constituting the part of the search space for which $Sa$ is responsible are dynamically and incrementally assigned by $Sa$'s parent agent (a network agent) and are stored in $ResponsibledNodes$. The dynamic assignment and distribution of a search space is presented in depth in the following section (section 4.2).

As in a forward-search algorithm, $Sa$ has a set of candidate nodes to expand $Frontier$ and a set of expanded nodes $Expanded$. In each iteration, $Sa$ executes the search algorithm which starts by selecting a node $n$ from $Frontier$ to expand. If $n$ is the destination, goal verification procedure (GVP) is initiated (further explained in section 4.3), and $Sa$ continues its execution until the destination is verified or a better path to the destination is found. Otherwise, $Sa$ generates the child nodes of $n$. To generate each child $n'$, it is necessary to know the cost for moving from $n$ to $n'$. We call the moving from $n$ to $n'$ an action $a(n, n')$. If $n'$ belongs to $Sa$'s part of the search space (i.e. in its $ResponsibledNodes$), $Sa$ stores $a(n, n')$ in its $ActionList$ to be queried for its cost. Otherwise, $Sa$ sends $a(n, n')$ to a search agent $Sa'$ that is responsible for $n'$ through the parent agent of $Sa$. Upon receiving $a(n, n')$, $Sa'$ discards $a(n, n')$ if it already knows the path to $n'$ with a better cost than through $n$; this prevents $Sa'$ from requesting for the cost to move from $n$ to $n'$, which is clearly in a non-optimal path. Otherwise, $Sa'$ adds $a(n, n')$ to its $ActionList$.

After node expansion, $Sa$ takes an action $a_q$ from its $ActionList$ to query for its cost. The cost of an action is determined by using information retrieved from resources associated with the action, and is computed using the cost function $f$ given as a part of MGPF. To obtain the cost of $a_q$, $Sa$ sends a request to a resource agent. A resource agent is capable of accessing a number of types of resources. Information about resource agents are provided to search agents as a part of their knowledge. Depending on the type of resource, $Sa$ chooses a resource agent to inquire. While the cost of $a_q$ is being retrieved, $a_q$ is moved from $ActionList$ to $PendingActionList$ where all the actions pending for their cost are stored. Retrieving an action cost is a non-blocking process. After sending the request to a resource agent, $Sa$ continues its execution. Once the necessary data is acquired, the resource agent sends it to $Sa$. This asynchronous mechanism for retrieving information enables search agents to perform other tasks while resources are being accessed, thus mitigates the latency.

After retrieving the cost of an action, $Sa$ processes received messages. When receiving a message containing the cost of an action $a(n, n')$, $Sa$ computes the cost of $n'$, $f(n')$ and removes $a(n, n')$ from $PendingActionList$. Based on $f(n')$, $Sa$ prunes unpromising nodes from $Frontier$ and updates $Expanded$. How the pruning is done depends on the actual algorithm (e.g. A*, uniform cost search). The algorithm is terminated when a verified solution is found or the entire search space has been explored. Naturally, we reach the end of a search space when all search agents have no node in their $Frontier$, no action in $ActionList$ and no pending action in $PendingActionList$.

### 4.2 The network agents

In this search model, we separate a search space based on the knowledge of the search space, in this case, the hierarchy information. A network agent is responsible for a set of hierarchy entities at a certain level of hierarchy. The role of a network agent is to manage the search process related to the hierarchy entities for which it is responsible. This management entails routing an action

to a relevant search agent, distributing workloads to search agents, assigning sub-hierarchy entities to other network agents to manage, creating new search agents and network agents when necessary, and handling communications among agents. Search space separation and assignment as well as workload distribution are done on-the-fly during the search in order to focus only on the parts of the search space relevant to a given pathfinding request. This process is triggered by the routing of actions, illustrated in Algorithm 1, to their relevant search agents, and progressively, it constructs an agent network tailored to a given request.

**Algorithm 1** action-routing-protocol($a\langle n, n'\rangle$)

```
1: hierarchy ← get the entire hierarchy of n'
2: if the executing agent Na is in charge of a hierarchy entity he in hierarchy then
3:    if he is the direct parent hierarchy entity (i.e. a leaf of HE) of n' then
4:       if Na has no search agent that is the agent responsible of n' then
5:          if Na has no search agents OR all search agents cannot take more respon-
            sibility then
6:             create a new search agent Sa and send a⟨n, n'⟩ to Sa
7:             set Na as the parent agent of Sa and Sa as a search agent of Na
8:          else
9:             select the search agent with the least responsibility and send it a⟨n, n'⟩
10:         end if
11:      else
12:         send a⟨n, n'⟩ to the search agent responsible
13:      end if
14:   else
15:      get he' from hierarchy where he' is a direct child hierarchy entity of he
16:      if Na has no child agents OR all child agents cannot take more responsibility
         then
17:         create a network agent Na' and make Na' responsible for he'
18:         set Na as the parent agent of Na' and Na' as a child agent of Na
19:      else
20:         assign he' to Na' where Na' is a child agent of Na with the least responsi-
            bility
21:      end if
22:      forward a⟨n, n'⟩ to Na'
23:   end if
24: else
25:   if Na's parent agent doesn't exist yet then
26:      create a network agent Na^p
27:      make Na^p responsible for the hierarchy entity that is the direct parent of Na's
         hierarchy entity(ies)
28:      set Na^p as the parent of Na and Na as the child of Na^p
29:   end if
30:   forward the request a⟨n, n'⟩ to the parent agent Na^p
31: end if
```

During node expansion, when a search agent $Sa$ discovers an action $a(n, n')$ where $n'$ is not under its responsibility, $Sa$ sends $a(n, n')$ to its parent agent $Na$. However, if it is the beginning of the search, which is an exceptional case where there are no network agents yet, the initial search agent $Sa^o$ creates the first network agent $Na$ to route $a(n, n')$. $Na$ becomes the agent responsible for the hierarchy entity in which $n$ is directly located. Upon receiving $a(n, n')$, $Na$ executes the action routing protocol (Algorithm 1) to find the search agent responsible for $n'$.

We separate a search space based on the hierarchy, so a search agent $Sa^{n'}$ that is responsible for $n'$, if it exists, is under the management of a network agent that is responsible for the hierarchy entity in which $n'$ is *directly* located. $Na$ uses the hierarchy information of $n'$ to guide the search (Algo:1 - L:1). The hierarchy of $n'$ is an ascending ordered list of hierarchy entities in which $n'$ is located. For instance, in an airport, the hierarchy of $n'$ can be *Zone 1-Terminal 2-Airport* where $n'$ is located directly under Zone 1. Zone 1 is a sub-hierarchy entity of Terminal 2, which is in turn a sub-hierarchy entity of Airport.

If $n'$ is not a part of $Na$'s search space (i.e. not located in one of $Na$'s hierarchy entities), $Na$ passes the control to its parent $Na^p$, if $Na^p$ already exists, to do the routing (Algo:1 - L:25-30). If $Na^p$ does not exist yet, $Na$ creates $Na^p$ to take charge of a hierarchy entity in which all $Na$'s hierarchy entities are located and forward the request to $Na^p$. For example, suppose $Na$'s hierarchy entities are Zone 1 and Zone 2 of Terminal 1; as $Na$'s parent, $Na^p$ takes charge of all the search processes in Terminal 1. Then, $Na^p$ takes over the routing operation and executes the action routing protocol. On the other hand, if $n'$ is located in one of $Na$'s hierarchy entities, denoted by $he^{Na}$, $Na$ takes control of the routing process (Algo:1 - L:2-23). This implies one of the two possibilities - (1) $n'$ is *directly* under $he^{Na}$ (i.e. $he^{Na}$ is a leaf of $HE$) or (2) $n'$ is *indirectly* under $he^{Na}$ (i.e. $n'$ is under a leaf, which in turns is under $he^{Na}$).

In the case of (1), the relevant search agent $Sa^{n'}$ should be under the management of $Na$. In such case, $a(n, n')$ is sent to $Sa^{n'}$ if $Sa^{n'}$ already exists (Algo:1 - L:12), and the routing operation of $a(n, n')$ is finished. If $n'$ has not been assigned to any search agent (i.e. $Sa^{n'}$ does not exist), $Na$ selects a search agent under its management that has the *least responsibility* to take charge of $n'$ (Algo:1 - L:9). To determine the responsibility of a search agent, we take into account its current workload, which is the number of nodes in its $Frontier$, and number of nodes for which it is responsible $ResponsibledNodes$. The workload indicates the current tasks that a search agent has to execute, and the number nodes in $ResponsibledNodes$ indicates the amount of potential tasks that it may have to do. The potential tasks include requesting the cost of actions, processing update messages and pruning. Using both criteria to measure responsibility enables us to assign more work to a search agent that is more likely to become idle (i.e. having few current tasks), and also preventing assignments to the ones that might potentially be occupied (i.e. responsible for many nodes). However, when all the search agents of $Na$ reach the *responsibility limit*, a new search agent is created to take charge of $n'$ (Algo:1 - L:5-7). The reason for introducing respon-

sibility limit is to distribute workloads among the search agents exploring the same part of the search space. This is essential when the part of the search space of a network agent is large. The responsibility limit is determined according to two factors: the computational resources available and the search space. If the computational resources are limited, the responsibility limit should be high to reduce the number of search agents. This configuration, however, may affect the efficiency when working with a large graph. Otherwise, the limit should be low, resulting in more search agents exploring in parallel.

In the case of (2), $Sa^{n'}$, if it exists, is under the management of one of the direct or indirect child agents of $Na$. $Na$ forwards $a(n, n')$ to its child agent $Na'$ (Algo:1 - L:15-22). $Na'$ is the direct child agent of $Na$ and is the agent responsible for a hierarchy entity $he'$ where $he'$ is a direct sub-hierarchy entity of $he^{Na}$ and $n'$ is located directly or indirectly under $he'$. If $Na'$ does not exist, $he'$ is assigned to a child agent with the least responsibility (Algo:1 - L:20). The responsibility of a network agent is measured by the sum of the number of nodes under the hierarchy entity(ies) for which it is responsible. We employ such indicator because the number of nodes determine the number potential tasks such as routing and other communications a network agent has to handle. Each network agent has a responsibility limit that is the maximum number of nodes it should handle. This limit is determined by the computational resources available. Setting the limit low results in having more network agents, but this would avoid problems such as communication bottlenecks. If all child agents of $Na$ reach the responsibility limit, $he'$ is assigned to a new network agent and $a(n, n')$ is forwarded to the new agent, which will continue the routing.

### 4.3 Goal verification procedure, termination and optimality

**Goal verification procedure** When a destination node $n_d$ is expanded, the expanding search agent $Sa^d$ initiates the GVP. The objective is to determine whether a found path leading to $n_d$ is a minimum-cost path. This process is necessary because each search agent does not possess global knowledge of the search state. The verification is conducted in a distributed manner by each search agent. A path is a minimum-cost path only if all search agents reach a consensus concerning its validity. For each search agent, a path to $n_d$ is optimal if there exists no node $n$ where $f(n) < f(n_d)$. To verify this property, each search agent performs the following verification:

- If there is any node $n$ in $Frontier$ where $f(n) < f(n_d)$, the path is not verified.
- If there are actions in $ActionList$ or $PendingActionList$, the path is not verified. The cost of those actions are still unknown, so it is impossible to determine the cost of the nodes to which those actions lead.

To start this procedure, $Sa^d$ sends a goal verification request to its parent agent $Na^d$. $Na^d$ propagates the request by forwarding the request to its other children and its parent agent. The receiving parent agent repeats the propagation process.

In this way, every search agent will receive the request from its parent agent. Each search agent keeps verifying the found path leading to $n_d$ until the path is verified or a better path to $n_d$ is found.

- If the found path is verified by an agent, the agent sends the response to its parent. Responses are sent back in the opposite direction of the one in which the request is propagated. Therefore, eventually, $Sa^d$ receives all the responses directly from $Na^d$, who receives responses from its parent and other children.
- If a better path to $n_d$ is found, the expanding agent initiates another GVP to replace the previous one.

Besides determining the validity of a path, GVP also enables search agents to filter nodes and actions. When the found path is under location verification of an agent, the knowledge about the found destination such as its cost is used to discard unpromising nodes from $Frontier$ and actions from $ActionList$.

**Termination** A search execution is terminated when a path verified by GVP is found or when the entire search space has been explored. Naturally, the end of a search space is reached when all search agents have no nodes in $Frontiers$, no actions in $ActionList$ and no pending actions in $PendingActionList$.

**Optimality** GVP enables us to determine whether a path has a minimum-cost. However, whether a path is optimal depends on the actual algorithm and the cost function(s) that it uses. For instance, in uniform cost search, the cost of a node is the cost from the start node to the node, which guarantees optimality. In such case, a path verified by GVP is an optimal path. However, for a greedy algorithm, a path verified by GVP is a minimum-cost path based on the algorithm's cost function, but not necessarily an optimal path.

The algorithm terminates by finding a minimum-cost path if one exists, assuming the following properties:

- The search space is finite.
- All messages arrive at their destinations.
- For every request for the cost of an action, we get a response.
- All operations take a finite amount of time.

## 5 Experimental evaluation

Our experiments were conducted on a 2.4GHz Intel Core i7 laptop with 16GB of RAM. We used 2 types of requests: the start and destination nodes are (1) in the same hierarchy entity (same hierarchy entity request) and (2) in different hierarchy entities (inter-hierarchy entity request). The principle of our approach is that it acquires information from various resources. As a result, we introduced simulated latency in accessing resources (1, 5, and 9 milliseconds). 1, 5 and 9

milliseconds (ms) are the maximum latency of each respective case. For instance, in the 5 ms case, we generate latency values between 0 to 5 ms. We applied our collaborative search model in uniform cost search (UC), creating a collaborative uniform cost algorithm (CUC). In our experiments, we compare CUC with UC. The choice of UC for our experiments is motivated by the fact UC is independent of any domain-specific or case-based heuristics. Consequently, the impacts of the collaborative search model on UC's performance can be accurately observed.

In the first experiment, we used both algorithms to solve the two types of requests on the same environment, abstracted as graph 1. Graph 1 has a 4-level depth hierarchy (1 hierarchy entity at first level, 10 at second, 100 at third, 1000 at fourth), 10000 locations and 10000 CPS entities. Fig. 3 demonstrates time efficiency in (%) gained by using CUC compared to UC to solve the two types of requests. The results show that for requests of type (1), UC is more efficient than CUC when there is no latency. This is because in type (1) requests, the start and the destination are under the same hierarchy entity containing approximately 100 nodes. In both algorithms, the solution was quickly found, but CUC takes more time because there are overheads for creating the network agent and managing communication as well as workload distribution. However, these overheads become negligible when latency is present. CUC starts to outperform UC from around 1 ms of latency. For type (2) requests, which involve multiple hierarchy entities, CUC performs better even without latency. The reason is that our model is based on concurrent agents exploring different parts of a search space (i.e. nodes under relevant hierarchy entities), which leads agents to discover the destination quicker. With latency, CUC is remarkably more efficient, reaching over 90% of time efficiency gains in the case of 9 ms latency. Regarding node expansion, in our model, each search agent has only partial knowledge of the search process, so it selects nodes to be expanded based on its limited knowledge. This may lead to expansion of costly or unpromising nodes. Despite such limitation, the results of our experiment, depicted in Fig. 4, suggest that CUC expands approximately the same number of nodes for type (1) requests and less for type (2). This is thanks to agent collaboration and the GVP. Collaborative and concurrent search leads to rapid discovery of the destination, irrespective of its optimality. Once the destination is found, the GVP is initiated, informing all search agents about the destination. While the destination is under local verification (i.e. verifying the properties described in section 4.3), search agents use knowledge about the found destination to filter unpromising nodes and actions (i.e. having higher cost than the found destination).

In the second experiment, we compared CUC with UC over three different graph structures of the same size, namely graph 1, 2 and 3. Graph 2 has the same hierarchy structure and location distribution as graph 1. The only difference is that in graph 2, there is only one connection between 2 locations and 1 exit point for each hierarchy entity, while there are 3 connections and 2 exit points in graph 1. Graph 3 has a 3-level depth hierarchy (1 hierarchy entity at first level, 10 at second, 100 at third), 10000 nodes (locations) and 10000 CPS entities. Fig. 5 illustrates time efficiency (in percentage) gained by using CUC compared to

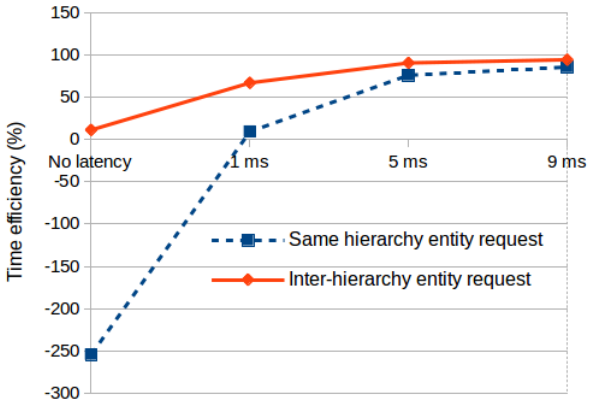


**Fig. 3.** Run time efficiency of collaborative uniform cost compared to uniform cost

UC on different graph structures. CUC performs best on graph 1 because there are more connections between nodes and more exits to other hierarchy entities. Our collaborative search model does not use a predefined method to separate a search space, but it dynamically and progressively distributes the search space among agents based on the structure of the space, in this case the hierarchy, during the search process. More exits to different hierarchy entities allow agents to reach more parts of the graphs faster, and thus finding the destination faster; more connections to other nodes also lead to a more efficient search since more nodes can be explored by concurrent agents. In graph 2 and 3, each node has only one neighbor node and a hierarchy entity has only one exit. With such connectivity, the algorithm takes more time to find the exits to enable agents to explore different parts of the graph. Between graph 2 and 3, CUC works better on graph 2 because locations are more distributed in graph 2. In graph 3, each 1000 locations are grouped under the same hierarchy entity, and there is only one exit from each hierarchy entity. In such case, the algorithm can only distribute the workloads, which are nodes under the same hierarchy entity, and has to expand many nodes to find an exit allowing spreading of the search to other hierarchy entities. This reason coupled with the overheads for communications and agent network management makes UC outperform CUC in graph 2 and 3 when there is no latency.

In the third experiment, we compared the two algorithms on three different graphs of the same structure and connectivity, but different sizes to examine the scalability. Graph 1, as described previously, has 10000 nodes; graph 4 has 4096 nodes, and graph 5 has 1296 nodes. Fig. 6 shows the order of growth of both algorithms over the three graphs. Regardless latency, CUC outperforms UC as the graph size grows due to the concurrent graph exploration by collaborative agents. Remarkably, CUC scales much better in function of latency.

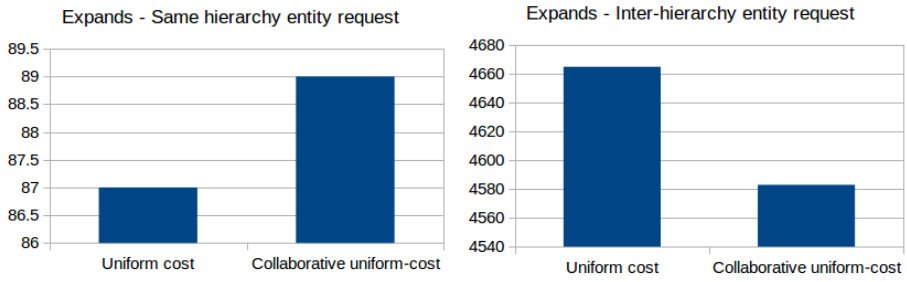


**Fig. 4.** Comparison of expands between collaborative uniform cost and uniform cost

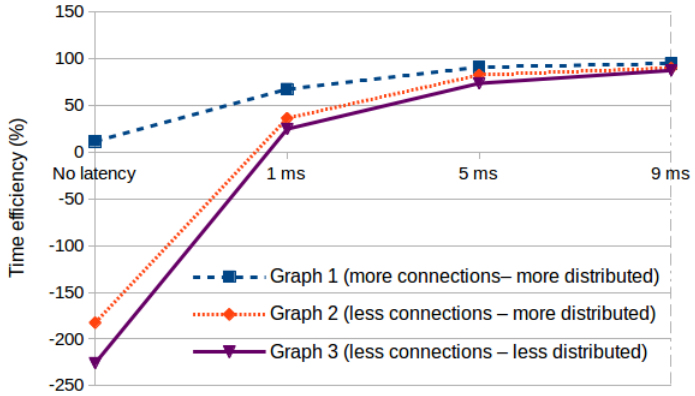


**Fig. 5.** Run time efficiency of collaborative uniform cost compared to uniform cost over different graph structures

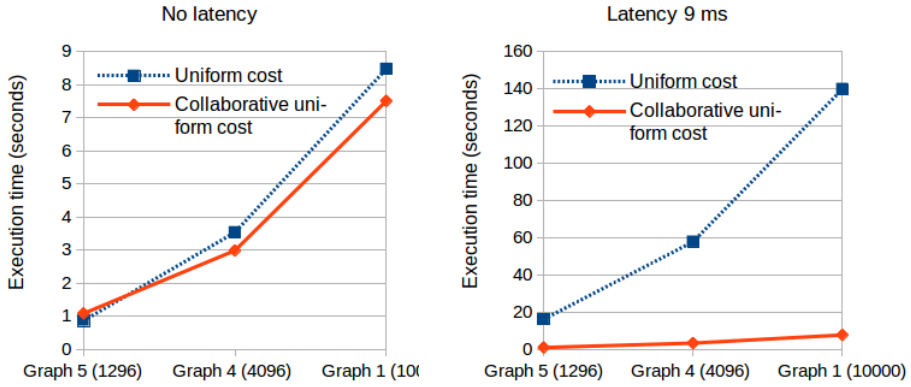


**Fig. 6.** Time comparison between collaborative uniform cost and uniform cost

These experimental results show the impact of graph structures, request types and latency on the efficiency of search algorithms. These factors can be used to choose a search algorithm suitable for each given MGPF problem.

## 6 Conclusion

This paper describes an approach to solve MGPF in ubiquitous environments by exploiting data from various resources from CPS entities to data sources on the Web. The overview of the approach was provided. We proposed a collaborative multi-agent search model that can be applied to forward-search algorithms to improve the efficiency and handle latency issue resulting from resource accesses. The model is based on agents searching collaboratively towards a shared goal. Such collaboration is enabled and facilitated by an agent network constructed by exploiting the structure and knowledge of a search space. While the collaborative search model is applicable to forward-search algorithms in general, we presented a specific example by applying it to uniform cost search. We used this example to demonstrate a concrete application of the model and to evaluate its efficiency. The results showed that the collaborative algorithm improves search efficiency in most cases, and scales better in function of latency and graph size.

In this paper, we focus on pre-trip planning. In ubiquitous environments, CPS entities are mobile and often changing their states. An activity for satisfying a goal takes a certain amount of time, during which CPS entities may be changing. Consequently, an optimal pre-planned path may lose its optimality over time. This necessitates en-route planning to keep refining the initial path according to the current state of the environment. In future work, we plan to extend our search model to support en-route planning. An agent network constructed during each search is tailored to that particular search. The network agents can be extended to support monitoring of CPS entities and resources under their coverage to detect mobility and changes. Such knowledge will then be taken in account to adapt the path on-the-fly while users are traveling.

## References

1. Algfoor, Z.A., Sunar, M.S., Kolivand, H.: A comprehensive study on pathfinding techniques for robotics and video games. International Journal of Computer Games Technology 2015, 7 (2015)
2. Botea, A., Bouzy, B., Buro, M., Bauckhage, C., Nau, D.: Pathfinding in games. In: Dagstuhl Follow-Ups. vol. 6. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik (2013)
3. Botea, A., Müller, M., Schaeffer, J.: Near optimal hierarchical path-finding. Journal of game development 1(1), 7–28 (2004)
4. Ciortea, A., Zimmermann, A., Boissier, O., Florea, A.M.: Towards a social and ubiquitous web: A model for socio-technical networks. In: 2015 IEEE/WIC/ACM International Conference on Web Intelligence and Intelligent Agent Technology (WI-IAT). vol. 1, pp. 461–468 (Dec 2015)

5. Codognet, P.: Multi-Goal Path-Finding for Autonomous Agents in Virtual Worlds, pp. 23–30. Springer US, Boston, MA (2003)
6. Denzinger, J.: Conflict handling in collaborative search. Conflicting agents pp. 251–278 (2002)
7. Ghallab, M., Nau, D., Traverso, P.: Automated Planning and Acting. Cambridge University Press (2016)
8. Graham, R., McCabe, H., Sheridan, S.: Pathfinding in computer games. The ITB Journal 4(2), 6 (2015)
9. Hart, P.E., Nilsson, N.J., Raphael, B.: A formal basis for the heuristic determination of minimum cost paths. IEEE Transactions on Systems Science and Cybernetics 4(2), 100–107 (July 1968)
10. Kishimoto, A., Fukunaga, A.S., Botea, A., et al.: Scalable, parallel best-first search for optimal sequential planning. In: ICAPS (2009)
11. Koenig, S., Likhachev, M., Furcy, D.: Lifelong planning A*. Artificial Intelligence 155(1-2), 93–146 (2004)
12. Lim, K.L., Yeong, L.S., Ch'ng, S.I., Seng, K.P., Ang, L.M.: Uninformed multi-goal pathfinding on grid maps. In: 2014 International Conference on Information Science, Electronics and Electrical Engineering. vol. 3, pp. 1552–1556 (April 2014)
13. Matai, R., Singh, S.P., Mittal, M.L.: Traveling salesman problem: An overview of applications, formulations, and solution approaches. Traveling Salesman Problem, Theory and Applications pp. 1–24 (2010)
14. Nissim, R., Brafman, R.I.: Multi-agent A* for parallel and distributed systems. In: Proceedings of the 11th International Conference on Autonomous Agents and Multiagent Systems-Volume 3. pp. 1265–1266 (2012)
15. Rabin, S.: A* speed optimizations. Game Programming Gems 1, 272–287 (2000)
16. Standley, T., Korf, R.: Complete algorithms for cooperative pathfinding problems. In: IJCAI. pp. 668–673. Citeseer (2011)
17. Standley, T.S.: Finding optimal solutions to cooperative pathfinding problems. In: AAAI. vol. 1, pp. 28–29 (2010)
18. Stentz, A.: Optimal and efficient path planning for partially-known environments. In: Robotics and Automation, 1994. Proceedings., 1994 IEEE International Conference on. pp. 3310–3317. IEEE (1994)
19. Vrakas, D., Refanidis, I., Vlahavas, I.: Parallel planning via the distribution of operators. Journal of Experimental & Theoretical Artificial Intelligence 13(3), 211–226 (2001)
20. Wang, K.H.C., Botea, A.: Fast and memory-efficient multi-agent pathfinding. In: ICAPS. pp. 380–387 (2008)
21. Werner, M.: Selection and ordering of points-of-interest in large-scale indoor navigation systems. In: 2011 IEEE 35th Annual Computer Software and Applications Conference. pp. 504–509 (July 2011)