Spatially Distributed Multiagent Path Planning

Christopher Wilt
Department of Computer Science
University of New Hampshire
Durham, NH 03824 USA
wilt at *cs.unh.edu*

Adi Botea
IBM Research
Dublin, Ireland
adibotea at *ie.ibm.com*

Abstract

Multiagent path planning is important in a variety of fields, ranging from games to robotics and warehouse management. Although centralized control in the joint action space can provide optimal plans, this often is computationally infeasible. Decoupled planning is much more scalable. Traditional decoupled approaches perform a unit-centric decomposition, replacing a multi-agent search with a series of single-agent searches, one for each mobile unit.

We introduce an orthogonal, significantly different approach, following a spatial distribution that partitions a map into high-contention, bottleneck areas and low-contention areas. Local agents called controllers are in charge with one local area each, routing mobile units in their corresponding area. Distributing the knowledge across the map, each controller can observe only the state of its own area. Adjacent controllers can communicate to negotiate the transfer of mobile units.

We evaluate our implemented algorithm, SDP, on real game maps with a mixture of larger areas and narrow, bottleneck gateways. The results demonstrate that spatially distributed planning can have substantial benefits in terms of makespan quality and computation speed.

Introduction

Multi-agent path planning is an important problem in a variety of fields, ranging from robotics to games. The problem is relevant to managing warehouse inventory (Wurman, D'Andrea, and Mountz 2008), where mobile robots in charge with transporting warehouse goods must coordinate with one another. In commercial games, mobile units have to navigate in a realistic fashion over a shared map.

We focus on cooperative path planning, a popular formulation among the multiple available flavours of the multi-agent path planning problem. Mobile units within a shared environment are interested in avoiding deadlocks and collisions with other units. The goal of each unit is to reach its target position, starting from a given initial position.

One way to optimally solve such problems is to centralize control. Centralized methods quickly become impractical as the number of mobile units grows. Recent years have seen a significant progress with optimal multiagent pathfinding (Standley 2010; Sharon et al. 2012), but the scalability remains a bottleneck in practice.

Decoupled suboptimal approaches are more practical. A popular strategy is to decompose the search problem in a unit-centric fashion, computing paths individually for each agent (Silver 2005; Wang and Botea 2008; 2011). Splitting the computation based on map decompositon is orthogonal to unit-centric decomposition. Work in multi-agent pathfinding based on this idea includes Ryan's (2008) research. As we argue in the related work section, there are methods that take steps towards combining unit-centric decomposition and map-based decomposition (Sturtevant and Buro 2006; Wang and Botea 2011).

Multi-agent pathfinding can be solved suboptimally in polynomial time (Kornhauser, Miller, and Spirakis 1984; Röger and Helmert 2012), but the practical performance of the available complete method is an open question.

In this paper, we focus on problems where spatial contention is highly localized, with many units having to visit some bottleneck areas. Unless such high-contention areas are handled efficiently, jams can quickly build up, with negative effects on the solving time, the quality of plans, and even the ability to solve the problem.

We introduce the Spatially Distributed Multiagent Planner (SDP), an algorithm that takes a map decomposition approach and distributes the computation across the space. SDP spawns agents called *controllers*[1] to manage various areas of the map. This allows some controllers to specialize in high-contention areas, whereas others are specific to low-contention areas. Adjacent controllers communicate with a message-exchange system, used to agree on transferring a mobile unit from one area to another. The routing of mobile units inside each area is solved locally, independently from the rest of the map. In particular, this allows handling each high-contention area separately from the rest of the problem, with a focus on smooth navigation at bottleneck points. The simple, limited communication interface between controllers allows combining heterogeneous local solvers, each of them suited to the specific properties (e.g., topology) of the local area at hand.

[1] We deliberately avoid using the term "agent", which could be ambiguous here. Traditionally, it is the mobile units that are called agents in multi-agent path planning.

To our knowledge, SDP is one of the first distributed algorithms for cooperative multi-agent path planning. Existing approaches, including algorithms in the "decoupled" category, rely on full map knowledge being available in a centralised way. Global knowledge, however, might not always be available, for reasons including privacy, limited communication, and limited sensing abilities, as discussed by Yokoo et al. (1998). Thus, in a distributed framework, the knowledge is distributed across controllers. Controllers know their own state, but they cannot directly observe the state of other controllers.

SDP is a suboptimal algorithm. As discussed in the paper, the way it implements the controllers does not preserve the completeness. SDP is evaluated empirically against recent, state-of-the-art multiagent pathfinding algorithms, such as MAPP (Wang and Botea 2011) and Parallel Push and Swap (Sajid, Luna, and Bekris 2012). We demonstrate the effectiveness of our ideas using real game maps with a mixture of open areas and narrow gateways.

The rest of the paper is structured as follows. Related work is reviewed next. We introduce a basic framework for spatial distribution, after which we present our algorithm SDP. This is followed by an empirical evaluation. The last part includes concluding remarks and future work ideas.

Related Work

The main focus of this section is work on cooperative multi-agent pathfinding. Relevant work in related problems, such as other multi-agent pathfinding formulations, and single-agent pathfinding is also included.

We start with suboptimal decoupled methods for cooperative pathfinding. Optimal methods have been discussed briefly in the introduction. Cooperative A* and Windowed Hierarchical Cooperative A* (WHCA*) (Silver 2005) reduce a problem to a series of single-agent 3D searches, where the third dimension is the time. A spatio-temporal reservation table marks the locations already reserved by other units, as a mechanism to avoid collisions. Sturtevant and Buro (2006) have extended WHCA*, combining the method with map abstraction. Abstraction constructs a multi-level map representation, with cliques at a lower level treated at single nodes at the next level (Sturtevant and Buro 2005). Abstraction improves the computation of individual paths. For example, restricting the search for a path to a narrow corridor, as opposed to the entire map, can improve performance. Our low-contention areas use a local planner based on Cooperative A*, as discussed later in this paper.

The MAPP algorithm (Wang and Botea 2011) assigns priorities to units, and computes an individual path for each unit. When resolving conflicts at runtime, low priority units move out of the way of higher-priority units. MAPP is complete for a class of problems called Slidable, and a few available extensions of that class. Besides being a unit-centric decoupled method, MAPP implements specialized rules for traversing single-width tunnels. This could be viewed as a step towards distributing the computation across the map.

As the name suggests, the Push and Swap algorithm (Luna and Bekris 2011) is a combination of two strategies. In a push step, units attempt to move towards the goal. When pushing is not possible, a swap macro-action is started, with the end result of swapping the positions of two units. Push and swap is complete on instances with at least two unoccupied vertices on the map graph. Parallel push and swap (Sajid, Luna, and Bekris 2012) is an extension that improves the quality of solutions, improving the ability to run multiple actions in parallel.

Flow Annotation Replanning (FAR) (Wang and Botea 2008) imposes unidirectional travel on top of an initially undirected gridmap. The travel direction alternates across the rows (and columns), covering the grid with criss-crossing virtual "road lanes". Topology-specific additional rules, applied locally, preserve the connectivity across the map. In terms of map abstraction, this boils down to removing some directed edges from a graph, as opposed to splitting a map into subgraphs. FAR computes a path for each unit independently, in a unit-centric decomposition. Conflicts, such as cycles, are addressed at runtime. A common idea with our work is restricting the traffic flow by ignoring some edges and imposing a travel direction on others. The flow inside SDP high-contention areas is restricted as shown later in this paper. We take advantage of the specific topology of high-contention areas, ensuring that they are free from local deadlocks and head-to-head collisions.

Ryan's (2008) abstraction approach partitions a map into specific structures, such as stacks, rings and cliques. For many maps, such as games gridmaps, it might be difficult to obtain a partitioning that would allow an effective solving process. Similarities with our work include the idea of partitioning a map, and the view that solutions need to include building blocks such as transferring units across components, or having units cross a component. Differences include our distributed approach, the actual way of partitioning a map, and the actual way to govern routing inside each area. Our motivation is to separate high-contention areas from low-contention areas.

Many of the previously surveyed methods employ best-first search methods, such as A*, on various search spaces, ranging from centralized multi-agent search to single-agent search. There are notable alternative approaches, involving different types of search. Bouzy (2013) introduces Monte-Carlo Fork Search (MCFS). Previous Monte-Carlo tree search methods have been very successful in domains such as Go. Yu and LaValle (2012) model multi-agent pathfinding as a multiflow optimisation problem, which is given as input to an off-the-shelf optimisation engine.

Map abstraction is popular in related, *single-agent* search problems, such as Sokoban (Junghanns and Schaeffer 2001; Botea, Müller, and Schaeffer 2002) and single-agent pathfinding (with a single mobile unit). Many methods in the later category implement various forms of map decomposition (Botea, Müller, and Schaeffer 2004; Sturtevant and Buro 2005; Björnsson and Halldórsson 2006; Harabor and Botea 2008; Yap et al. 2011).

In our approach, entry points to each area to be visited by a mobile unit can be seen as a sequence of subgoals. Subgoals and related concepts, such as landmarks, have successfully been used in domain-independent planning and other search problems (e.g., (Korf 1987; Koehler and Hoffmann

2000; Porteous, Sebastia, and Hoffmann 2001)).

Distributed path planning has been investigated in a setting where units are allowed to simultaneously share an edge (Lim and Rus 2012). This is significantly different from our domain. Finding any solution is easy, boiling down to putting together individual paths computed independently. Thus, research has focused on optimal planning, under a congestion-related assumption that the time to traverse an edge depends on the number of units using the edge.

De Mot et al. (2002) address a different problem where multiple units with limited local sensing seek to arrive at the *same* target, and mobile units are allowed to share the same vertex. Finding a legal solution is simple, and work has focused on optimality. In that work, the term *spatial distribution* refers to the way agents position across the map, to communicate efficiently, and to gather as much information as possible about the map, while still allowing the units to exploit the knowledge that was collectively gathered.

Spatial Distribution

Spatial distribution is based on a partitioning of the map graph G into $k \geq 1$ components $C_1, \ldots C_k$. To achieve this, the vertex set V is partitioned into disjoint subsets $V_1, \ldots V_k$. Edges connecting two vertices in component C_i belong to that component. Edges connecting two components are *transition edges*. Each component is controlled by an agent called a controller. As there is a 1-to-1 mapping between components and controllers, we will use these names interchangeably, when the clarity is not affected.

The knowledge is distributed across controllers. A controller knows its own state (topology and configuration of mobile units) but it does not know the state of other controllers. A controller is responsible for all routing inside its component. Furthermore, two adjacent controllers negotiate the transfer of units from one component to another.

We distinguish between *crossing moves* and *internal routing moves*. A crossing move is a single step of a unit, along a transition edge, from one component to another. Internal routing moves are performed inside a component without interacting with the rest of the map. Internal routing can serve for three purposes: have contained units reach local targets (*target macros*); create the conditions to transfer a unit to an adjacent controller (i.e., take the unit to the border); and create the conditions to accept a unit from an adjacent controller (i.e., make available a given position by the border). We call a *transfer macro* a collection of moves including: a possibly empty internal-routing sequence ψ to bring a unit to the border of a controller c; a possibly empty internal-routing sequence ψ' to make room at the border of a controller c'; and the corresponding crossing move from c to c'.

Observation 1. *For any valid solution to a multi-agent path planning problem, there exists a valid solution, obtained through a possibly empty reordering of the moves, that is a sequence of transfer macros and target macros.*

To transfer a unit u from c to c', the two controllers need to agree on a transfer place (i.e., a transition edge (l, l')) and a transfer time. Controller c checks if it can bring u to a border location l, after which it sends a message to c', asking whether c' can accept unit u at an adjacent location l'. When controller c' can accept the unit, a corresponding transfer macro π is generated. In general, there can be many possible transfer macros π, depending on the position by the border where the transfer is made, and the actual sequences ψ and ψ'. In the discussion on the completeness presented later in this section, we take into account the following condition:

Condition 1. *Instantiating a macro into actual moves (e.g., ψ, ψ' and the crossing move for a transfer macro) systematically enumerates possible combinations.*

Multi-agent plans can be sequential (totally ordered) or they can allow moves in parallel. A theoretical discussion on complenetess is simpler under a sequential-plan assumption. This is why, in this section, we restrict our attention to sequential plans, even though our implemented algorithm, presented later, allows parallel movements.

Consider a synchronous framework, where controllers take turns at taking the initiative (i.e., performing a target macro internally or initiating communication to perform a transfer macro). Thus, assume a total ordering among controllers, and assume that a token is passed around, in a circle, to indicate the current controller to take the initiative. When controller c gets the token, it can choose between three options: directly pass the token further, or perform a target macro, or initiate a communication for a transfer macro.

Observation 2. *The ability to directly pass the token to the next controller ensures that arbitrary (valid) sequences of transfer macros and target macros can be generated.*

In the overall search space, the options of a controller with the token (i.e., choosing one of the available macros, or choosing to pass the token directly) create new branches. Furthermore, various low-level realizations of a given macro create different branches as well. In such a resulting search space, one needs to recognize deadends, goal states, and cycles (repetitions of the global configuration of the mobile units along a given exploration branch).

When trying to perform a given macro, a controller runs into a deadend if no low-level instantiation of the macro is found, despite searching systematically at the local level of the component(s) at hand. Controllers can identify a global goal state through a communication mechanism where each controller reports if its local state is a local goal. Repetitions of the global unit configuration along a given branch can be identified in a similar, but slightly more involved way. Each local component maintains a history of its local positions at relevant time moments such as the initial moment and the times after a macro is generated. Given a current time moment, each component can report a list of previous time moments with a local state identical to the current one. If the lists reported by the components have a non-empty intersection, a global repetition of the unit configuration has been detected. In particular, this implies that passing the token directly all the way through in a completed circle results in a duplicated global state.

Theorem 1. *When Condition 1 holds, the generic distributed framework is complete.*

Proof Sketch: Observation 1 states that restricting the attention to sequences of target macros and transfer macros

preserves the completeness. Observation 2 and Condition 1 state that the distributed framework allows to generate arbitrary valid sequences of low-level moves that can be seen, at the higher abstraction level, as a sequence of transfer macros and target macros. These facts combined prove the result.

Mobile Unit Interface

To keep the previous completeness discussion as straightforward as possible, we assumed the entire system was governed by controller agents. No heuristic guidance has been discussed so far, such as indicating, for a given unit and a given host controller, what should be the next controller where to transfer the unit. A high-level path, with a sequence of components to visit all the way to the goal, could be a valuable heuristic. One way to integrate such heuristic guidance is to assume that mobile units store a high-level path, being able to specify the next controller to visit.

Such information is provided on request to a hosting controller, so that the controller knows what adjacent controller to contact for a transfer. Obviously, when the hosting controller is the last one on the abstract path at hand, the controller attempts to locally route the unit to its target destination. To maintain completeness, units may need to backtrack on their high-level paths.

Controller Agent Interface

While the precise details of how various controller agents work will differ, all controllers need a core set of abilities. First, a controller must be able to determine whether it can successfully route a mobile unit arriving at a particular vertex and time, and simultaneously honor all previous commitments it has made. A controller must also be able to commit to accepting mobile units at a particular vertex and time. Last, in order to allow for completeness, controllers should be able to undo the effects of accepting a mobile unit, which will allow backtracking. The mobile unit will have to find either a different arrival time, or a different high-level path.

The SDP Algorithm

SDP, our implemented algorithm, is based on the generic framework outlined in the previous section. It trades away completeness for efficiency, and it uses high-level individual paths as heuristic guidance. SDP makes use of two types of controllers, corresponding to high-contention and low-contention areas.

Each unit has a high-level path consisting of a sequence of components to be visited by that unit. We opted for a simple and fast-to-code implementation of high-level path computation: run an A* search on the original map, ignoring all other units. Then, remove all unnecessary details, such as actual low-level moves. If desired, this step can be sped up with abstraction: treat each component traversal as a single action (macro step), and run A* in that abstracted, smaller graph. Both alternatives could respect the limited-knowledge principle adopted in the distributed approach. For example, in abstracted search, give a controller an entry vertex together with its g cost, and request a set of outgoing vertices with their g costs. The controller computes this internally, similarly to abstraction methods for single-agent pathfinding mentioned in the related work section. As mentioned earlier, in general, more than one high-level path might have to be considered to ensure completeness. Being incomplete, SDP considers only one high-level path per unit.

Algorithm 1 Main processing loop in SDP

```
1: t = 0                                        ▷ time step in solution
2: while True do
3:     for each controller c do
4:         processTurn(c,t)
5:         if global goal then
6:             return solution
7:     t++
```

Algorithm 1 outlines the main processing loop in SDP. In the method processTurn, a controller can perform internal routing as part of goal and transfer macros, communicate with adjacent controllers, and transfer units. The rest of this section shows how high-contention and low-contention controllers are built and how they work, routing mobile units internally and transferring units as needed.

High-contention controllers

A high-contention component contains a narrow passageway, called the *corridor center area*, together with a surrounding *buffering area*, used as a waiting place while either entering or exiting the corridor.

Examples of a corridor center area L include a few locations in a row forming a narrow gate, or a few contiguous locations forming a loading area. An area L can be seen as an intermediate disjunctive goal for the units that need to cross it. For example, in order to cross a gate, one has to reach at least one (but not necessarily all) of the gate locations.

Identifying corridor center areas on the map. We identify corridor centers with a number of patterns, an example of which is illustrated in the left part of Figure 1. This pattern identifies vertical corridors that are 2 cells wide and less than 8 cells long, centered at the cells marked with a "C". There are similar patterns for finding horizontal corridors, and corridors of different widths and lengths. In our implementation, we detected corridors that were 2, 3 and 4 cells wide.

The pattern is moved around the map like a sliding window, to identify corridors and their corresponding center cells. After all of the patterns have been run on the map, adjacent corridor centers are merged to form a single longer corridor center. Figure 1 is used as a running example for the construction of a high-contention component after the corridor center area has been identified.

Building a high-contention area around a corridor center set. Each (merged) corridor center identified as shown is used to create a high-contention area around it. The first step of this process is to impose direction restrictions on corridor center cells. For vertical corridors, the cells on the

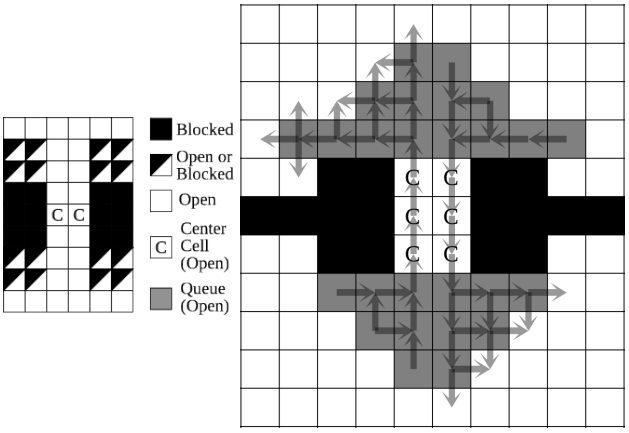

Figure 1: A sample pattern for finding corridors for grids (left), and an example of a high-contention component (right). In both figures, the corridor centers are marked with a C. Buffering areas are in gray. Permissible routes in the high-contention component are denoted by arrows.

left allow an upwards-oriented flow, whereas the cells on the right are for travelling downwards.[2] The separation line between the two directions is used as shown next.

The next step is to add cells to create the buffering area. This is done once for each direction of travel by initializing a breadth first search at the corridor centers for that direction, and expanding the first N cells that are on the same side of the separation line, besides the cells inside the gateway tunnel (we used $N = 13$). The breadth-first search depth of each node, which we call the *BFS depth*, is recorded. After both breadth-first searches have finished, the high-contention controller's vertexes have been identified.

The last step is to verify that creating the high-contention area will not cause problems for the map. If it causes either low-contention area to be disconnected, the high-contention area is removed. Likewise, if there are short paths around the high-contention area, it is easy to get around the high-contention area, making it unnecessary.

Internal routing in a high-contention component. Internal routing takes units from the border of an incoming buffer to the corresponding corridor center and then further to the border of the corresponding outgoing buffer. A few simple rules ensure that no deadlocks occur, and that all units eventually succeed in crossing a high-contention area. Units that have not yet crossed the corridor center are always routed to an empty vertex with a lower BFS depth, giving priority to the unit that has been in the controller longer to ensure that no unit is starved. Ties are broken in an arbitrary but consistent manner. There are no head-to-head collisions (i.e., units trying to travel in opposite directions along the same edge e). The existence of such collisions would imply that each of the two end nodes of e would have a smaller BFS depth than the other, which is a contradiction.

Units that have already crossed the corridor center are always routed to a cell with a greater BFS depth (on the other side of the corridor center). When heading away from the corridor center, some cells are adjacent to a neighbor low-contention controller. If this is the case, the high-contention controller attempts to pass the unit on to the next controller. If the next controller is unable to accept the transfer request, the unit is pushed further into the outgoing buffer of the current controller. If the unit has reached the end of the buffer, or if the deeper cells are already occupied, the unit is instructed to wait. At the next time step, the high-contention controller repeats the attempt to pass the unit to the next controller or push the unit deeper into the exit buffer.

Algorithm 2 High Contention Can Accept Mobile Unit

```
1:  function CANACCEPTUNIT(unit u, location l, time t)
2:      t_c ← current time
3:      κ_p ← previous acceptance commitments   ▷ Triples (u', l', t'), with t_c ≤ t'
4:      κ ← κ_p ∪ {(u, l, t)}
5:      σ ← current unit configuration in this HCA
6:      t_s ← t_c                               ▷ Initialize simulation time
7:      while κ ≠ ∅ do
8:          for all (u', l', t') ∈ κ with t' ≤ t_s do
9:              if l' is occupied in σ then
10:                 return False
11:             Remove (u', l', t') from κ
12:             Add unit u' at location l' in σ
13:         Update σ        ▷ Relaxed simulation of internal routing one time step ahead (see text)
14:         t_s ← t_s + 1
15:     return True
```

As mentioned earlier, controllers, including high-contention controllers, can receive requests to accept a given unit at a given time. The decision whether to accept a given unit u, at a location l and a future time t boils down to the availability of a free location in the incoming buffer at that time, without impacting previously made acceptance commitments. This decision procedure is outlined in Algorithm 2. The check involves simulating ahead the controller's internal routing, starting from its current configuration and taking into account previous acceptance commitments. The simulation uses the relaxing assumption that units can leave the current controller at the time they make it to the border of the outgoing buffer. In reality, it is possible that a mobile unit may be stuck in the high-contention area waiting for the next controller to accept the transfer. Even if one or several mobile units need to wait until they exit the high-contention area, the controller might still be able to accept new incoming units, unless the controller area is already full. We found that optimistically assuming that mobile units would leave on time worked well, unless the density of mobile units in an instance was too high. This optimistic assumption, which affects SDP's completeness, is discussed further in the experiments section.

[2]For odd widths, ties are broken in favor of the direction with heavier expected demand, based on initial high-level plans of units. We found that breaking ties randomly also works well.

Low-contention controllers

After building high-contention controllers, every remaining maximal contiguous area is a low-contention component.

A low-contention controller is responsible for routing mobile units across the low-contention component, to the next high-contention component, or to local target locations. Each low-contention controller uses Cooperative A* (Silver 2005), restricted to the low-contention area at hand, and adapted as discussed later in this section. Cooperative A* performs individual searches in a 3D space, with time being the third dimension. The space/time positions of units with already planned paths are treated as obstacles when computing the paths of the remaining units.

Algorithm 3 Low Contention Can Accept Mobile Unit

```
1: function CANACCEPTUNIT(unit u, location l, time t)
2:     Run A* with start (l, t), 3D search space, 3D disjunctive goal test (see text)
3:     if no path found then
4:         return False
5:     else
6:         Store the path for future use
7:         return True
```

When a low-contention controller receives a request to accept a unit, it runs an A* search in the 3D space to compute a path from the entry location to either the target location, or the next controller. If a path is found, the request is accepted (Algorithm 3), and the path is used to route the unit.

If the path computation fails or all of the replies from the next controller are negative, the current low-contention controller rejects the acceptance request at hand. Otherwise, it accepts the incoming unit and the 3D trajectory of the computed path is added to the set of obstacles to be considered in subsequent 3D searches in the low-contention area at hand.

Standard Cooperative A* turned out to require a number adaptations to work in our problem. To understand why, consider the main differences between standard multi-agent path planning and routing inside a low-contention area. In the standard case, mobile units have a specific target location, and any arrival time is acceptable in terms of solution validity. We extend a standard goal state both spatially and temporally. In our problem, there could exist multiple communication points between two adjacent components. As a result, we use a more permissive definition of a "destination location", allowing disjunctive destinations. A disjunctive destination contains locations adjacent to the current controller owned by the next controller. In addition, not all arrival times at a next component are acceptable. Thus, testing whether a state in the 3D search space is a goal is extended with a temporal condition. Assume that the spatial component (x, y) of a 3D state (x, y, t) is a disjunctive destination of the current search. To consider it a goal state, the accepting controller has to agree that it can accept the incoming unit at the location (x, y) with an arrival time of t.

Low-contention controllers are also responsible for routing mobile units to their final goals. This is achieved by bringing the mobile unit to its goal location, and at every time step, re-planning to bring the mobile unit back to its goal. Most of the time, the plan is trivial, consisting of one "sit" move. Unlike in Cooperative A*, mobile units at their goal do not reserve the use of the goal location. Thus, other units can plan their paths through the goal of a unit u. When this happens, u's re-planning will force it to temporarily vacate the goal position and return at a subsequent time when no other unit reserved that location.

Empirical Evaluation

We ran experiments comparing SDP against state-of-the-art multiagent path planning algorithms. The authors of Parallel Push and Swap (Sajid, Luna, and Bekris 2012) and the authors of MAPP (Wang and Botea 2011) kindly gave us copies of their code to use for experiments. These programs are written in C++. As mentioned in the related work section, a remarkable property of both these benchmark algorithms is that they offer completeness guarantees for well-specified classes of problems. SDP is implemented in Java. Experiments were carried out on a Macbook Pro with a 2.26 Intel Core 2 Duo and 4GB of RAM running OSX 10.8.3.

For evaluation domains, we consider 27 different maps from Dragon Age: Origins (Sturtevant 2012). All algorithms allow 4 way motion, and mobile units must completely vacate a node before another mobile unit may enter the vacated node.

Mobile units were assigned random start locations that were outside of the high contention areas. Goals were also assigned randomly, with the additional restriction that goal locations were neither inside a high contention area, nor among the 90 closest cells to a high contention area. This typically produced a buffer that was 2 cells wide. We imposed this restriction because our implemented high-contention controllers do not support goals or start locations within or next to the high-contention areas. While addressing this efficiently might be challenging, there are many scenarios where such a restriction makes sense. For example, in real life, bridges do not host parking lots or petrol stations. The narrow space available on bridges should be used for passing, and all locations that could act as vehicle destinations (e.g., parking lots or petrol stations) should be placed where more room is available.

Given a map and a number of mobile units, let the *density* be the number of mobile units divided by N, the number of traversable cells on the map. For each map, the density is initialised to 0.25%, and gradually increased in increments of 0.25% of N at a time. For every combination of a map and a density, we conducted 10 trials, each trial consisting of a different collection of mobile unit start and goal locations, and averaged the results of the 10 runs. Among all these discrete increments, the max number of mobile units considered for a map corresponds to the case where SDP completed all ten runs without halting prematurely. The implemented SDP halts prematurely when a high-contention controller optimistically accepts a unit transfer request, but it is unable to accept the transfer at the agreed time.

The total number of problem instances considered in our experiments, built as shown earlier, is 3,500. Our first experiments evaluate how SDP performs compared to MAPP

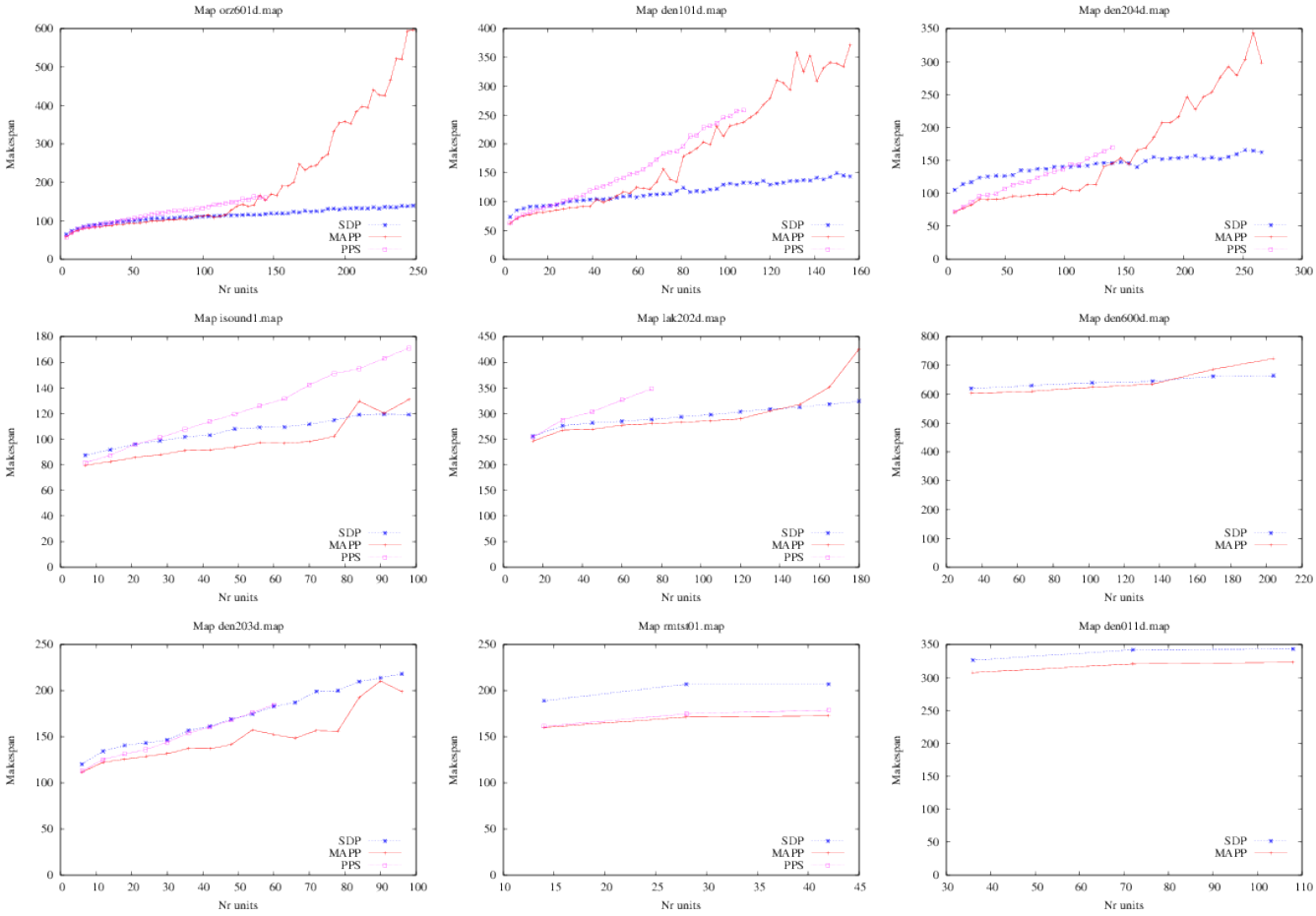

Figure 2: Solution quality measured in makespan vs number of mobile units on a number of Dragon Age: Origins maps.

and PP&S as we change the number of mobile units. On all maps, we enforced a timeout of 5 minutes per instance. PP&S ran into this limit on some problems. So did MAPP in a few cases. Any algorithm that failed to find solutions to all 10 instances of a given agent density was dropped from the higher mobile unit densities.

The results of this comparison can be seen in Figure 2. We show a representative subset of all maps. First, many data points for PP&S (and a few for MAPP) are missing, due to failures for the reasons previously mentioned. Each row shows one of the three tendencies we have observed in the data. At the top, SDP computes solutions with a significantly better makespan. In easier instances, where the mobile unit density is low, SDP is initially outperformed by MAPP and, more seldom, by PP&S. This is due to the fact that SDP imposes restrictions on how the mobile units go through doorways, which can increase the makespan in simple instances. Eventually SDP starts dominating. The differences increase as the instances grow in density. The middle row illustrates a similar tendency (e.g., SDP eventually dominating), but the makespan differences are smaller.

As the number of mobile units increases, MAPP begins to build plans with significantly larger makespans. Mobile units are routed towards their destinations, but sometimes mobile units will get stuck in a traffic jam near a doorway, and stop moving. Eventually, MAPP resolves the traffic jam by sliding the mobile units in question around to allow the higher priority units to get through, but doing this is time consuming, producing plans with a long makespan. As the number of mobile units increases, the likelihood of this occurring in any given doorway increases. The same general trend can be seen in PP&S, for those cases where PP&S scales up. In contrast, SDP deploys dedicated controllers (the high-contention controllers) to govern the high demand cells at and near each doorway, which keeps the makespan performance more stable as we add additional mobile units.

The bottom row in Figure 2 illustrates cases where MAPP provides better or comparable makespans for all densities considered. In this category, the differences between MAPP and SDP are stable and relatively low. The difference can be as high as 33% of SDP's makespan (map hrt201d). It is around 5% or less for maps such as den505d and den005d. An exception from the summary just outlined is map orz201d. For two out of 24 densities, SDP makespans

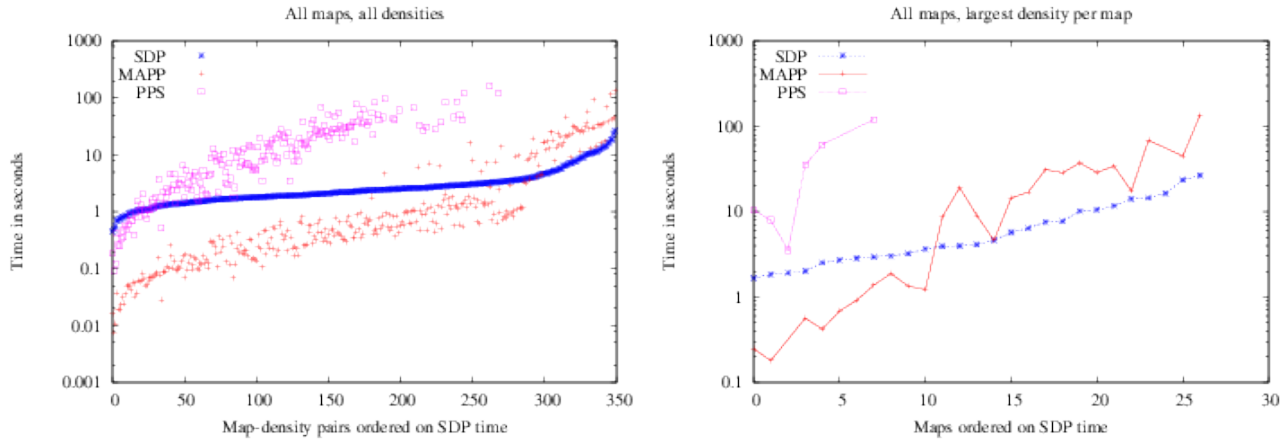

Figure 3: Running times in seconds. Given a map-density combination, the results are averaged over the 10 trials considered.

are 3.17 and 3.45 times as large as PP&S makespans. For one density, the SDP makespan is 3.57 times as large as the MAPP makespan. For all other densities on this map, the makespans are largely comparable, with SDP being slightly better than PP&S and slightly worse than MAPP.

At a closer analysis we observed that, on maps such as hrt201d, most of the area ends up as one enormous room. There are doorways that would allow a partitioning, but they were not identified by our implemented detectors. If this happens, SDP largely reduces to Cooperative A* run on (almost) the entire map, with a corresponding performance degradation. Alternative low-contention controllers based on other, more recent algorithms than Cooperative A* might be considered in principle. On the map den203d, a different problem arises. Some of the rooms for low-contention controllers are long and thin. Adding the buffer for a high-contention controller creates a new bottleneck separating two parts of the adjacent low-contention area, making path planning in the region inefficient.

In summary, each row in Figure 2 illustrates a distinct tendency observed in the makespan data. The separation line between these categories is informal to some extent. However, we note that the top row is representative for 9 out of 27 maps. The middle row corresponds to 6 maps. For three maps, no significant differences were observed between MAPP's and SDP's makespans.

As the number of mobile units further increases, the optimistic high-contention controllers we implemented sometimes made commitments that, in reality, they were unable to honor. This is a situation that we detected, but did not address in this work. Instead, our main focus was to study the impact that spatial distribution can have on the computation speed and the quality of plans. Figure 2 shows approximately how well we can expect SDP to scale currently.

Figure 3 summarizes the speed results, showing running times on a logaritmic scale. At the left, all map-density pairs used in experiments are taken into account. At the right, we show the running times for the highest density considered for each map. The speed data show two general trends. The first is that, on the easier instances, where SDP requires a few seconds or less, SDP is generally slightly slower than MAPP. The second trend, however, is that on the more difficult instances SDP has a substantial advantage. PP&S is significantly slower than both other programs in most cases.

Conclusion

Existing approaches to cooperative multi-agent path finding, including centralized search and decoupled techniques, assume centralized problem knowledge. We have introduced a distributed framework for multi-agent path planning, based on distributing the knowledge and the control across a map. Our aim with this framework was to set a basis for building actual algorithms based on spatial distribution.

We have developped such an algorithm, SDP, aimed at handling efficiently maps with a mixture of narrow bottlenecks and larger areas. In multi-agent pathfinding, the existence of small areas with a high spatial contention can greatly increase the difficulty of a problem. Solving high-contention areas separately from the rest of the problem can be key in solving a problem efficiently.

In SDP, we created specialized controllers in charge with one map area each. Controllers exchange messages to agree on transferring mobile units. Thus, the architecture is simple and modular. Any controller that implements the simple messaging interface could in principle be plugged in. Our results show that even a simple, incomplete implementation of the high-contention and the low-contention controllers could be effective, solving instances with bottleneck areas significantly more efficiently than state-of-the-art methods.

In future work, we plan to analyze the connections between decoupled and distributed multi-agent path planning. Existing decoupled algorithms might be possible to adapt to a distributed framework. In addition, we believe that combining spatial decomposition with unit-centric decomposition can be taken further, to obtain more scalable solvers.

References

Björnsson, Y., and Halldórsson, K. 2006. Improved heuristics for optimal path-finding on game maps. In *Proceedings of the Conference on Artificial Intelligence and Interactive Digital Entertainment AIIDE-06*, 9–14.

Botea, A.; Müller, M.; and Schaeffer, J. 2002. Using Abstraction for Planning in Sokoban. In Schaeffer, J.; Müller, M.; and Björnsson, Y., eds., *Proceedings of the 3rd International Conference on Computers and Games CG-02*, volume 2883 of *Lecture Notes in Artificial Intelligence*, 360–375. Edmonton, Canada: Springer.

Botea, A.; Müller, M.; and Schaeffer, J. 2004. Near optimal hierarchical path-finding. *Journal of Game Development* 1:7–28.

Bouzy, B. 2013. Monte-Carlo Fork Search for Cooperative Path-finding. In *International Joint Conferences on Artificial Intelligence (IJCAI) workshop on Computer Games*.

De Mot, J.; Kulkarni, V.; Gentry, S.; and Feron, E. 2002. Spatial Distribution Results for Efficient Multi-Agent Navigation. In *Proceedings of the 41st IEEE Conference on Decision and Control*, 3776–3780.

Harabor, D., and Botea, A. 2008. Hierarchical Path Planning for Multi-Size Agents in Heterogeneous Environments. In *Proceedings of the IEEE Symposium on Computational Intelligence and Games CIG-08*, 258–265.

Junghanns, A., and Schaeffer, J. 2001. Sokoban: Enhancing general single-agent search methods using domain knowledge. *Artificial Intelligence* 129(1-2):219–251.

Koehler, J., and Hoffmann, J. 2000. On reasonable and forced goal orderings and their use in an agenda-driven planning algorithm. *Journal of Artificial Intelligence Research* 12:338–386.

Korf, R. E. 1987. Planning as search: a quantitative approach. *Artificial Intelligence* 33(1):65–88.

Kornhauser, D.; Miller, G. L.; and Spirakis, P. G. 1984. Coordinating pebble motion on graphs, the diameter of permutation groups, and applications. In *25th Annual Symposium on Foundations of Computer Science (FOCS)*, 241–250.

Lim, S., and Rus, D. 2012. Stochastic distributed multiagent planning and applications to traffic. In *IEEE International Conference on Robotics and Automation (ICRA)*, 2873–2879.

Luna, R., and Bekris, K. E. 2011. Push and swap: Fast cooperative path-finding with completeness guarantees. In *International Joint Conferences on Artificial Intelligence (IJCAI)*, 294–300.

Porteous, J.; Sebastia, L.; and Hoffmann, J. 2001. On the extraction, ordering, and usage of landmarks in planning. In *Recent Advances in AI Planning. 6th European Conference on Planning (ECP'01)*, 37–48.

Röger, G., and Helmert, M. 2012. Non-optimal multi-agent pathfinding is solved (since 1984). In *International Symposium on Combinatorial Search (SOCS)*.

Ryan, M. R. K. 2008. Exploiting subgraph structure in multi-robot path planning. *Journal of Artificial Intelligence Research (JAIR)* 31:497–542.

Sajid, Q.; Luna, R.; and Bekris, K. E. 2012. Multi-agent pathfinding with simultaneous execution of single-agent primitives. In *International Symposium on Combinatorial Search (SOCS)*.

Sharon, G.; Stern, R.; Felner, A.; and Sturtevant, N. R. 2012. Conflict-based search for optimal multi-agent path finding. In *Proceedings of the 26th National Conference on Artificial intelligence (AAAI)*.

Silver, D. 2005. Cooperative pathfinding. In *AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment (AIIDE)*, 117–122.

Standley, T. S. 2010. Finding optimal solutions to cooperative pathfinding problems. In *Proceedings of the 24th National Conference on Artificial Intelligence (AAAI)*.

Sturtevant, N., and Buro, M. 2005. Partial Pathfinding Using Map Abstraction and Refinement. In *Proceedings of the 20th National Conference on Artificial Intelligence (AAAI)*, 47–52.

Sturtevant, N. R., and Buro, M. 2006. Improving collaborative pathfinding using map abstraction. In *AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment (AIIDE)*, 80–85.

Sturtevant, N. 2012. Benchmarks for grid-based pathfinding. *Transactions on Computational Intelligence and Artificial Intelligence in Games* 4(2):144 – 148.

Wang, K.-H. C., and Botea, A. 2008. Fast and memory-efficient multi-agent pathfinding. In *Proceedings of the 20th International Conference on Automated Planning and Scheduling (ICAPS)*, 380–387.

Wang, K.-H. C., and Botea, A. 2011. Mapp: a scalable multi-agent path planning algorithm with tractability and completeness guarantees. *Journal of Artificial Intelligence Research (JAIR)* 42:55–90.

Wurman, P. R.; D'Andrea, R.; and Mountz, M. 2008. Coordinating hundreds of cooperative, autonomous vehicles in warehouses. *AI Magazine* 29(1):9–20.

Yap, P.; Burch, N.; Holte, R. C.; and Schaeffer, J. 2011. Block A*: Database-Driven Search with Applications in Any-Angle Path-Planning. In *Proceedings of the 25th National Conference on Artificial Intelligence AAAI*.

Yokoo, M.; Durfee, E. H.; Ishida, T.; and Kuwabara, K. 1998. The distributed constraint satisfaction problem: Formalization and algorithms. *IEEE Transactions on Knowledge and Data Engineering* 10:673–685.

Yu, J., and LaValle, S. M. 2012. Planning optimal paths for multi-agent systems on graphs. *CoRR* abs/1204.3830.