Published in the Proceedings of the Sixth Annual Symposium on Combinatorial Search (SoCS), 2013

From Feasibility Tests to Path Planners for Multi-Agent Pathfinding

Athanasios Krontiris
Rutgers University
Piscataway, NJ 08854 USA
tdk.krontir@gmail.com

Ryan Luna
Rice University
Houston, TX 77005 USA
rluna@rice.edu

Kostas E. Bekris
Rutgers University
Piscataway, NJ 08854 USA
kostas.bekris@cs.rutgers.edu

Abstract

Multi-agent pathfinding is an important challenge that relates to combinatorial search and has many applications, such as warehouse management, robotics and computer games. Finding an optimal solution is NP-hard and raises scalability issues for optimal solvers. Interestingly, however, it takes linear time to check the feasibility of an instance. These linear-time feasibility tests can be extended to provide path planners but to the best of the authors' knowledge no such solver has been provided for general graphs. This work first describes a path planner that is inspired by a linear-time feasibility test for multi-agent pathfinding on general graphs. Initial experiments indicated reasonable scalability but worse path quality relative to existing suboptimal solutions. This led to the development of an algorithm that achieves both efficient running time and path quality relative to the alternatives and which finds a solution on available benchmarks. The paper outlines the relation of the final method to the feasibility tests and existing suboptimal planners. Experimental results evaluate the different algorithms, including an optimal solver.

Introduction

Multi-agent pathfinding requires the computation of paths for agents on a graph, where the agents move from initial to goal vertices while avoiding collisions. It has applications in warehouse management (Enright and Wurman 2011), space exploration (Luna, Oyama, and Bekris 2010), intelligent transportation (Dresner and Stone 2008), and computer games (Silver 2005; Wang and Botea 2008; Jansen and Sturtevant 2008).

Background: Providing an optimal solution for this problem is NP-hard. Optimality can be typically achieved by coupled methods, which consider all agents as a single composite system. These methods quickly become intractable as the number of agents increases. Recent solutions improve performance by searching in an iterative deepening manner (Sharon et al. 2011) or modifying state expansion using operator decomposition and segmenting instances into independent subproblems (Standley 2010; Standley and Korf 2011). Even these state-of-the-art optimal methods, however, can provide solutions only for up to tens of agents (e.g., 50 agents). Decoupled techniques are computationally efficient alternatives (Wang and Botea 2008; Jansen and Sturtevant 2008; Silver 2005; Sturtevant and Buro 2006; Masehian and Nejad 2010) that compute individual paths and resolve collisions as they arise. While they solve problems much faster, they are not complete. This paper focuses on algorithms that provide a solution if one exists and scale to thousands of agents with a sacrifice in path quality.

An encouraging result in algorithmic theory is that answering the feasibility question, i.e., whether a solution exists or not, can be answered efficiently. An early theoretical study (Kornhauser 1984), which refers to the challenge as the *"pebble motion on a graph"* problem (PMG), provided a polynomial time solution to the feasibility question. It built on top of earlier work, which dealt with the case of biconnected graphs with one agent fewer than vertices (Wilson 1974). More recent progress, however, has shown that the problem is solvable not just in polynomial but in linear time (Auletta et al. 1999; Goraly and Hassin 2010). In particular, a linear-time feasibility test was first provided for the "pebble motion on trees" problem (PMT) (Auletta et al. 1999). Recent work proved that the tree-based solution can be utilized to also provide a linear-time answer for general graphs with two more vertices than agents (Goraly and Hassin 2010). A recent report provides an equivalent approach for general graphs (Yu 2013). The permutation invariant case has also been addressed (Yu and LaValle 2012).

The results in algorithmic theory had not received much attention in the combinatorial search community until recently (Roger and Helmert 2012). Thus, researchers in this community developed independently polynomial-time, suboptimal but complete solvers for multi-agent pathfinding. Some of these combinatorial search methods focused on specific graph topologies, such as trees (Peasgood, Clark, and McPhee 2008), biconnected graphs with two more vertices than agents (Surynek 2009) or "slideable" grid-based problems (Wang and Botea 2011). Another method provides a polynomial-time solution for general graphs with two empty vertices (Luna and Bekris 2011b). The linear-time feasibility tests, however, can potentially form the foundations for efficient path planning solutions. This was the idea in related, recent work (Khorshid, Holte, and Sturtevant 2011), which proposed a path planner that corresponds to the feasibility test for the PMT problem (Auletta et al.

1999), although it was developed independently to it and inspired by other recent work on the subject (Masehian and Nejad 2009). To the best of the authors' knowledge no path planner corresponding to the linear-time feasibility test for general graphs (Goraly and Hassin 2010) has been provided before.

Contribution: This paper first outlines a path planner for multi-agent pathfinding based on the approach to transform a graph into a tree (Goraly and Hassin 2010) so as to utilize a polynomial path planner for trees (Khorshid, Holte, and Sturtevant 2011). The integration of the existing approaches left many choices to be decided during the development of the overall method so as to achieve an efficient implementation. Comparisons against optimal coupled planners (Standley and Korf 2011) and sub-optimal solutions (Luna and Bekris 2011a; Sajid, Luna, and Bekris 2012) showed that the resulting planner achieves scalability in terms of running time but results in low quality solutions. The paper identifies the reasons for the poor path quality and proposes a way to address it by utilizing search primitives used in existing sub-optimal solutions for graphs (Luna and Bekris 2011b). The result is a new algorithm that shares features of both the linear time feasibility tests and existing suboptimal solvers. The paper describes the relationship of the new algorithm with the existing search solutions and the operations of the feasibility tests. The final algorithm achieves improved path quality relative to the alternatives, as well as strong performance in terms of running time. Experiments are included that compare the different algorithms, including an optimal planner, in terms of running time and path quality.

Setup

Consider an undirected *graph* $G(V, E)$ and a set of *pebbles* $\mathcal{P}$. The number of vertices is $n = |V|$ and the number of pebbles is $k = |\mathcal{P}|$. An *assignment* $A : \mathcal{P} \rightarrow V$ is a function that places pebbles on vertices and is *valid* if it is injective, i.e., if it assigns pebbles to unique vertices:

$$\forall i, j \in \mathcal{P}, i \neq j : A[i] \in V, A[i] \neq A[j].$$

Unoccupied vertices on G given A will be called *holes* and will be denoted as the set $\mathcal{H}(G, A)$. The number of holes in G given A will be denoted as: $|\mathcal{H}(G, A)|$. A graph with no holes given an assignment will be called a *full graph*.

A *valid action* $\alpha(A_a, A_b)$ is a transition between assignments A_a and A_b so that only one pebble moves between neighboring vertices, i.e., $\exists\, i \in \mathcal{P}$ and $\forall j \in \mathcal{P}, j \neq i$:

$$A_a[j] = A_b[j],\ A_a[i] \neq A_b[i],\ (A_a[i], A_b[i]) \in E.$$

A *multi-pebble path* $\Pi = \{A_0, A_1, \ldots, A_{|\Pi|}\}$ is a sequence of valid assignments where for any two consecutive assignments A_i and A_{i+1} in Π there is a valid action $\alpha(A_i, A_{i+1})$.

Definition 1 (`PMG` **path planning with 2 holes)** *Given a graph $G(V, E)$, a set of pebbles $\mathcal{P}$, where $k \leq n - 2$, a start valid assignment A^S and a target valid assignment A^T, compute a multi-pebble path $\Pi = \{A^S, \ldots, A^T\}$.*

This work deals with the case $k \leq n - 2$ addressed in the linear-time feasibility test for graphs (Goraly and Hassin 2010) and in previous suboptimal solvers. The case of $k = n - 1$ can also be addressed based on a different feasibility test (Wilson 1974) but is not considered here.

A *branch vertex* $w \in V$ is a vertex that has more than 2 neighbors on G ($degree(w) > 2$). The variables q and r are used to denote neighbors of branch vertices. The notation $\pi(v, u)$ is used to describe the shortest path (sequence of vertices) between v and u in V, while $dist(v, u)$ denotes the length of this path. Moreover, the *visibility region* $v.seen$ of a vertex v given A is defined as follows:

$$v.seen = \{u \in V : \forall\, x \in \pi(v, u)/u, x \in \mathcal{H}(G, A)\}$$

i.e., the subset of the graph that can be connected to v with a path that is not blocked by any pebbles, including the vertices occupied by pebbles "visible" by v.

Given a node $v \in V_T$ of a tree $T(V_T, E_T)$, the notation T^v will be used to represent the forest that arises by deleting v and its incident edges from T. Moreover, T^v_u denotes the subtree in forest T^v that contains vertex u. The operation $T^v - T^v_u$ returns the set of subtrees from the forest T^v except the one that contains u.

Existing Work on Trees

This section first describes existing work on how it is possible to reason in a linear-time whether a problem is solvable on trees (Auletta et al. 1999) and a corresponding path planner (Khorshid, Holte, and Sturtevant 2011).

From PMT to PPT: The feasibility test (Auletta et al. 1999) reasons by first reducing the `PMT` problem into a pebble permutation on a tree (`PPT`) problem, where the pebbles have been moved to occupy the same vertices as in the target assignment A^T but not necessarily in the correct order. This reduction can be achieved in linear time: starting from the leaves of T, for each vertex that appears in the target assignment A^T, the method moves the closest pebble on that vertex. Define as $\hat{A}^S$, the permutation of the target assignment that is reachable by the start assignment with this process. For a connected T, this process will always succeed. The `PMT` problem is solvable only if the corresponding `PPT` problem is solvable.

Solving PPT instances: Given a `PPT` problem, it is possible to collect information in linear time that can be used to detect whether the target assignment is reachable from the start permutation. In particular, a procedure, referred to in this work as FIND_TREE_INFO, collects the following information for each vertex v that is occupied according to A^T:

1. $v.seen$;
2. the number of holes for each subtree of the forest T^v;
3. the closest branch vertex w on each sub-tree of T^v;

This information is valid as long as the pebbles are assigned to any permutation of the target assignment A^T. Starting again from the leaves of T the function propagates this information towards the interior of the tree and then outwards again until all vertices have informed their neighbors about these values. This information is useful to detect if two vertices occupied by pebbles in the target permutation A^T belong in the same "equivalence" class. Pebbles that occupy vertices in the same equivalence class can be swapped. The equivalence property between vertices is transitive in nature. For the `PPT` problem to be solvable, it has to be that for every $p \in \mathcal{P} : \hat{A}^S[p] \equiv A^T[p]$, i.e., there is a way for pebble p to swap with the pebble occupying its target vertex.

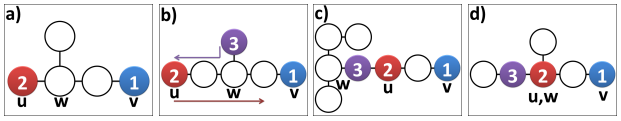

Figure 1: Criteria for the equivalence of vertices v and u: a,b) the cases of the first criterion; c) for criterion 2: $|\mathcal{H}(T^u - T^u_v, A^T)| \geq dist(w,u) + 2$; d) in criterion 3 one of the v, u vertices is a branch vertex.

Equivalence detection: The following criteria are used to decide the equivalence of two occupied vertices v and u:
- Vertices u and v are equivalent, if there is a branch vertex w ($degree(w) > 2$) so that any of the following is true:

- **Criterion 1)** $w \in \pi(v,u)/\{u,v\}$ and $|\mathcal{H}(T^w - T^w_v - T^w_u), A^T| \geq 1$. In this case, the pebbles on u and v can swap on a branch vertex that lies between them but not on u or v. See Fig. 1a) and 1b).
- **Criterion 2)** $u \in \pi(v,w)$ and $|\mathcal{H}(T^u - T^u_v), A^T| \geq dist(w,u) + 2$. This criterion checks for branch vertices that are in other subtrees of u relative to v. Similarly for $v \in \pi(u,w)$. See Fig. 1 c).
- **Criterion 3)** $w = v, |\mathcal{H}(T^v - T^v_u, A^T)| > 2$ and at least two trees from the forest T^v are not full. Similarly for $w = u$. See Fig. 1 d).

Checking these criteria can be achieved in constant time for each pair of neighboring pebbles according to $v.seen$ given the information collected from function FIND_TREE_INFO.

Existing Tree Planner: A polynomial-time path planner for solving multi-agent pathfinding problems on trees has already been proposed (Khorshid, Holte, and Sturtevant 2011). The algorithm starts from the leaves of the tree and works inward so as to find the first goal state that is not occupied by the correct agent. It moves this agent to its goal by swapping it with any agents along the path to the goal. This action will be repeated until all the agents will be on their goal states. In order for the algorithm to recognize if two agents can swap, similar criteria as in the feasibility test described above have been used. After a successful swap is completed, the algorithm restores the positions of all agents except the two that swapped their positions. This approach works on trees, as well as a class of problems on graphs called slideable.

A Path Planner For Graphs

Graph Conversion to Tree: A way to answer the feasibility question for the pebble motion on a graph (PMG) problem is to first transform the graph $G(V,E)$ into a tree $T(V_T, E_T)$ (Goraly and Hassin 2010), so that if the PMT problem is solvable on T then the PMG problem is solvable on G. It works by converting each maximal nontrivial 2-vertex-connected component $S \subset G$ (M2CC for short) into a star topology, which is a linear time procedure. This is achieved by connecting all vertices of the M2CC with a new virtual "trans-shipment" vertex s. In particular, the tree T is initiated so that $V_T = V$ and $E_T = E$. Then, for every $\mathcal{S} \subset G$, which is M2CC ($\mathcal{S}$ is non-trivial if it has more than one vertex):

1. Add a transshipment vertex s: $V_T = V_T \cup \{s\}$;
2. Remove all edges of $\mathcal{S}$: $\forall\, e \in E(\mathcal{S}) : E_T = E_t/\{e\}$;
3. Add edges with s: $\forall\, u \in V(\mathcal{S}) : E_T = E_T \cup \{(u,s)\}$.

Examples of such transformations are provided in Fig. 2. A constraint for the resulting PMT problem is that pebbles are not allowed to stop on trans-shipment vertices but only pass through them. Moreover, the tree T has weights: all edges connected with a trans-shipment vertex have a weight of $\frac{1}{2}$, while the remaining ones have a weight of 1.

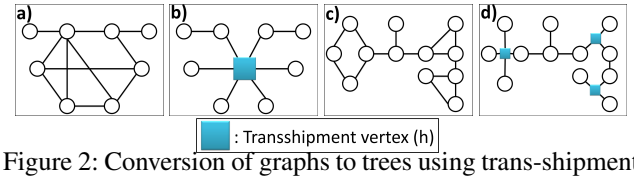

Figure 2: Conversion of graphs to trees using trans-shipment vertices. The graph from figure (a) will be transformed into the tree in figure (b), while the graph from figure (c) into (d).

Path Planner: Given the above description, it is possible to define a new path planning solution for general graphs by first incorporating a procedure similar to the work by (Goraly and Hassin 2010) and turn a PMG challenge into a PMT one, which can be addressed by a procedure equivalent to the existing solver for trees (Khorshid, Holte, and Sturtevant 2011). The actual algorithm for solving the PMT problem considered in this work follows closer the original feasibility test for trees (Auletta et al. 1999). The overall solution is described at a high-level in Algorithm 1.

Algorithm 1: PMG_VIA_PPT_SOLVER$(G, \mathcal{P}, A^S, A^T)$

```
1  Convert G(V,E) → T(V_T, E_T);
2  Π_T ← ∅;
3  Â ← REDUCE_PMT_TO_PPT(T, A^S, A^T, Π_T);
4  FIND_TREE_INFO(T, A^T);
5  forall the pebbles p ∈ P ordered based on A^T do
6      π ← COMPUTE_PATH(Â[p], A^T[p]);
7      forall the pebbles p' ∈ P along π do
8          (w, case) ← FIND_BRANCH(T, Â[p], Â[p']);
9          if w ≠ NULL then
10             SWAP(w, case, p, p', T, Π_T, Â);
11         else
12             return FAILURE;
13 Π_G ← CONVERT_PATH(Π_T, G, T);
14 return Π_G;
```

The first four lines of the path planner directly correspond to steps of the feasibility tests:

a) The graph G is converted into a tree T through the use of trans-shipment vertices (line 1).
b) The PMT problem is reduced to a PPT problem by moving the pebbles from A^S to a permutation $\hat{A}$ of the target A^T. The path planner stores the actions that move the pebbles from A^S to $\hat{A}$ in the multi-pebble path Π_T (lines 2-3).
c) The necessary info is collected by FIND_TREE_INFO and stored on the tree T (line 4).

The algorithm considers the pebbles in an order defined by A^T: pebbles with targets closer to leaves of T have priority over those at the interior of T (line 5). Given this ordering and the PPT problem, after a pebble has reached its target, it will not have to participate in a swap again.

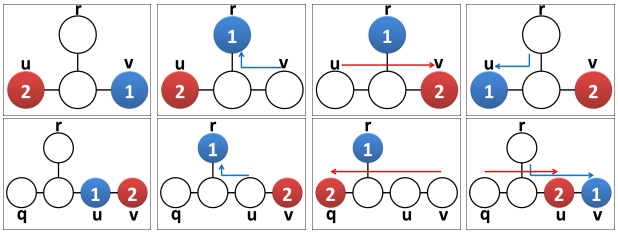

Figure 3: Illustration of the swap operations. Top: The branch vertex is between the two pebbles. Bottom: Both pebbles are on the same side of the branch vertex.

For each pebble p, the shortest path is computed that takes it from its current vertex $\hat{A}[p]$ to its target $A^T[p]$ (line 6). To solve the problem, all the pebbles p' along the path have to incrementally swap vertices with pebble p (line 7). For each pair p and p', the function FIND_BRANCH returns a branch w, which the pebbles can use to swap (line 8). It performs the same operation as the "equivalence detection" process of the tree feasibility algorithm and returns the criterion that corresponds to w (variable $case$), i.e., 1) w lies between $\hat{A}[p]$ and $\hat{A}[p']$; 2) w is the first branch along a subtree of $\hat{A}[p]$ not including $\hat{A}[p']$ - similarly for the symmetric case; 3) w is either $\hat{A}[p]$ or $\hat{A}[p']$. FIND_BRANCH utilizes the data computed by FIND_TREE_INFO. If the problem is solvable, a branch w exists and a swap will take place (lines 9-12).

Swap Operations: Depending on the case of w, SWAP creates different actions to achieve the switch between p and p' and stores them on Π_T. Only the first case is outlined below due to space limitations, which is indicative, however, of the other two cases.

In particular, when w lies between between vertices u and v occupied by p and p' the following steps are executed. It is first necessary to check if there is a subtree of w with a hole. If it exists, its root is cleared by finding the first empty vertex v_e on T^w and moving all the pebbles along the path $\pi(r, v_e)$ to free vertex r. Then the steps shown in Fig. 3 (top row) are executed. In the case that there is no tree of w with holes, a pebble from a full subtree T^w_q will have to move in one of the T^w_v or T^w_u subtrees so as to empty its root q. In this case, both pebbles are in the same sub-tree of T^w and the steps shown in Fig. 3 (bottom row) need to be performed. During SWAP, pebbles p and p' can tentatively move other pebbles away from the targets. At the end of the SWAP function there is a call to a REVERT function, which guarantees that all pebbles will return to the same vertices they occupied before SWAP, with the exception of p and p', which have now swapped vertices. Upon completion, the current assignment $\hat{A}$ is again a permutation of A^T as required in a PPT problem.

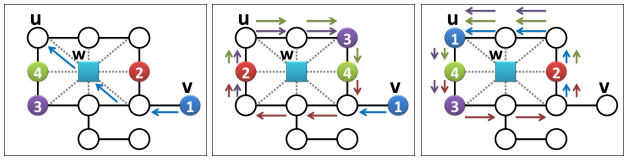

Figure 4: Cyclical rotations during path conversion so that pebble 1 will reach its goal u (w is a trans-shipment vertex).

Converting the Path: Up to line 12 algorithm PMG_VIA_PPT_SOLVER, has computed a path Π_T on T, which contains actions along trans-shipment vertices. The function CONVERT_PATH replaces such actions with alternatives that go through G (line 13). When a pebble moves from v to u via a trans-shipment vertex s, the function first checks if there is a pebble-free path between v and u on G. If there is, this movement can be achieved. Otherwise, a more complex process is implied by the feasibility algorithm for graphs (Goraly and Hassin 2010). Each trans-shipment vertex corresponds to an M2CC subgraph $\mathcal{S} \subset G$. Such a subgraph must contain a cycle $\mathcal{C} \subset \mathcal{S}$ connected to an external edge. Every action that goes from v to u via s can be replaced by a sequence of rotations on a cycle $\mathcal{C} \subset \mathcal{S}$ that contains v and u (Goraly and Hassin 2010). Fig. 4 provides an illustration of the type of rotational actions that arise when a solution goes through a trans-shipment vertex and needs to be converted to a path that goes through edges of the graph.

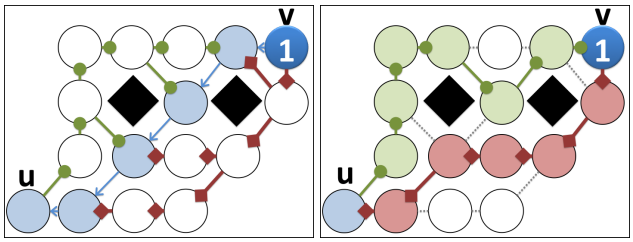

Figure 5: Computing a cycle on a maximally 2-edge-connected connected in order for pebble 1 to reach vertex u. (left) the case where the shortest path between u and v splits the component in two parts (right) the resulting cycle that can arise by the procedure outlined below.

The feasibility algorithm, however, did not require computing the cycle $\mathcal{C}$, which can have a significant impact on solution quality and running time. CONVERT_PATH implements the following: First the shortest path π is computed between v and u. Then, an attempt is made to find a second path π' that connects v and u and doesn't go through any intermediate vertex of π. If such a path is found, then the cycle $[\pi(u, v)|\pi'(v, u)]$ is returned. If not, the removal of the vertices along π has separated $\mathcal{S}$ into two disconnected components, as illustrated in Fig. 5. Then, the algorithm identifies the vertices along π reachable with an alternative path from v: $reach(v)$ and from u: $reach(u)$. Given that a cycle must exist, it has to be that there is a pair of vertices u' and v' along π so that: a) $u' \in reach(u)$, $v' \in reach(v)$ and b) $dist(u, u') > dist(u, v')$. Then it is possible to return the cycle: $[\pi'(u, u')|\pi(u', v)|\pi'(v, v')|\pi(v', u)]$. Overall, this is a linear time procedure.

Path quality is also affected by choices made when a trans-shipment vertex is a branch vertex for a swap. Fig. 6 shows the issue: pebble 1 on v wants to swap with pebble 2 on u. The algorithm finds the trans-shipment vertex w as a branch vertex. In Fig. 6a-c), the top left corner is selected as the intermediate stop r to be used during the swap resulting in a relatively high number of steps when that path is translated from T to G. A better choice for r is to find the closest vertex to v and u on $\mathcal{S}$ so that $r \notin \pi(v, u)$. Fig. 6d-f) show

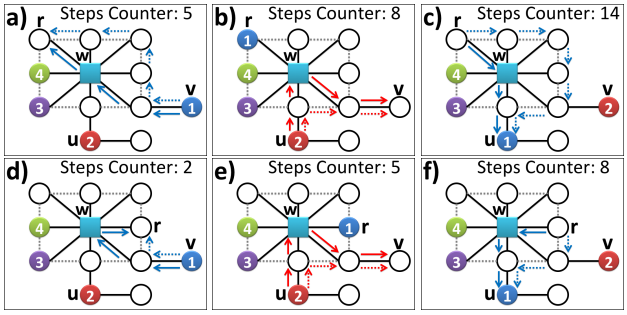

Figure 6: Choosing an intermediate node r during a swap around a trans-shipment vertex w impacts solution quality.

the steps if such a vertex was chosen. A similar situation arises when both pebbles are in the same sub-tree of T^w and two vertices must be found on $\mathcal{S}$. They should be the closest ones to the subtree that contains the pebbles.

Limitations: Initial experiments indicated that the above planner scales reasonably well but that the path quality achieved is not satisfactory. This can be attributed to the following:

- Maintaining the problem in PPT form requires the pebbles to backtrack and acquire the position of a target.
- Solving the problem first on T and then converting it on G often creates large rotations involving multiple pebbles along cycles $\mathcal{C}$ of maximal 2-vertex-connected components of G just to move a pebble between two vertices.

These aspects are direct results of the insights, which are critical in allowing a linear time feasibility test. Reasoning over a tree and bringing the pebbles back to targets means it is possible to use the data collected by FIND_TREE_INFO to quickly identify the branch w in FIND_BRANCH where p and p' can swap. It is interesting, however, to investigate if it is possible to achieve improved path quality, while still achieving competitive running time. Given the issues identified above, it is interesting to look into solutions that can identify if two pebbles can swap while they occupy general vertices on graphs, i.e., without a transformation into a tree and without requiring the pebbles to occupy specific vertices, for which precomputed data are available.

Search-based Solution with Improved Quality

Push and Swap approach: An approach that has been proposed to address similar challenges directly on graphs is the "Push and Swap" (PaS) algorithm (Luna and Bekris 2011b), which can be seen as a generalization of the existing path planner on trees (Khorshid, Holte, and Sturtevant 2011) for general graphs with two holes. Given the knowledge of the linear-time feasibility tests and the corresponding solver described in the previous section it is possible to define a new approach that does not suffer from the limitations of existing solutions. In particular, PaS considered all possible vertices of degree ≥ 2 as a potential branch where two pebbles can swap, which is unnecessary. The equivalence detection procedure shows it is sufficient to consider only branches in a local neighborhood around the pebbles. Furthermore, PaS imposed the following three requirements, which are similar in nature to the properties of PMG_VIA_PPT_SOLVER:

- Upon the completion of a swap, every pebble returns to its previous position and the two swapped pebbles return to their previous locations but they have switched vertices.
- Pebbles that have reached their targets cannot be moved by other pebbles unless a swap needs to use their vertex.
- Two pebbles can swap only after they occupy neighboring vertices. The feasibility algorithms show that a swap can be detected as soon as two pebbles are in the visibility region of one another.

The new PMG_SOLVER does not impose any of these requirements, which results in an approach that is simpler and achieves improved performance compared against PaS as illustrated in the section providing the experimental results.

Algorithm 2: SEARCH_SWAP(G, p, p', Π, A)

```
1  v ← A[p],  u ← A[p'];
2  W ← GET_BRANCHES(v, u);
3  forall the w ∈ W in order of distance from u do
4      Π_w ← ∅; Π_c ← ∅;
5      if !TEST_MOVE(G, Π_w, v, w) then
6          break;
7      neigh : neighbor of w closer to A[p'];
8      Π_w ← Π_w + MOVE(A[p'], neigh);
9      N ← GET_NEIGHBORS(w) − neigh;
10     Blocked ← {w, neigh};
11     forall the n ∈ N do
12         if TEST_CLEAR(G, Π_c, n, Blocked) then
13             Blocked ← Blocked + {n};
14     if there is no empty neighbor for w then
15         break;
16     else
17         r : an empty neighbor of w;
18         if there is only 1 empty neighbor for w then
19             if !TEST_MOVE(G, Π_c, w, neigh) then
20                 break;
21             q ← random(N − r);
22             Π_c ← Π_c + MOVE(q, r) +
               MOVE(A[p], w) + MOVE(A[p'], neigh);
23             if !TEST_CLEAR(G, Π_c, r, Blocked) then
24                 break;
25         else
26             q : another empty neighbor of w;
27     Π+ = Π_w + Π_c + SWAP_OPS2(w, neigh, r, q);
28     return TRUE;
29 return FALSE;
```

A New Search-based Swap Primitive: This work proposes a new primitive for simultaneously detecting the feasibility and performing a swap of two pebbles p and p' that are within their visibility region on a general graph with two holes. In other words, a method that does the job of both FIND_BRANCH and SWAP directly on a graph. The method does not try to bring the pebbles back to their previous vertices after the completion of swap but only to remove pebble p' from the shortest path of pebble p. This means it is no longer possible to utilize the information computed by FIND_TREE_INFO to detect if a specific branch can be used for the swap. This information needs to be computed by searching the graph given the latest pebble assignment.

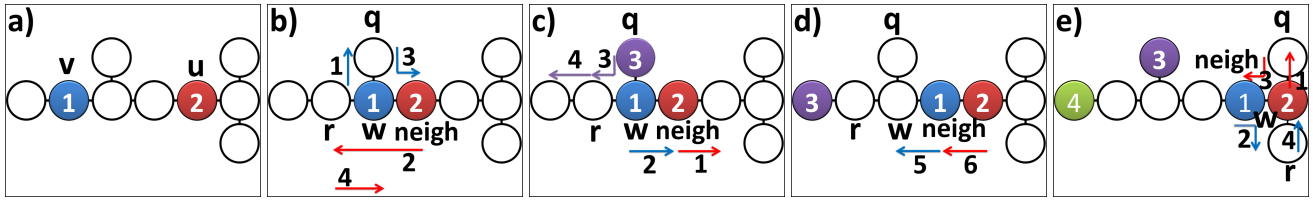

Figure 7: (a) Consider pebbles 1 and 2 that need to swap. (b) 1 moves to w and 2 occupies $neigh$; vertices r and q are the empty neighbors of w that help the swap. The arrows show SWAP_OPS2' steps. (c) In this case, pebble 3 blocks q and needs to be evacuated through r. (d) After 3 moves, pebbles 1 and 2 can perform the swap. (e) If 1 and 2 have considered the branches along their path and the swap cannot take place there, they check the closest branches outside their path.

Algorithm 2 describes the proposed SEARCH_SWAP primitive, which starts by considering the candidate branches W between and around vertices v and u, occupied by pebbles p and p' correspondingly (lines 1-2). The set W is the same set as the one identified by the equivalence detection procedure and includes all three cases. As an optimization, the vertices in W can be sorted based on the distance from pebble p', as the assumption is that p' lies along the shortest path of p and p needs to move towards this direction (line 3). The algorithm first tries to bring pebble p on the branch w and pebble p' on the adjacent vertex $neigh$ of w (lines 4-8), as shown in Fig. 7a-b). The only exception to this rule is if the vertex w is located further away from u relative to v, in which case p' is moved to w and p is moved to $neigh$ (not shown in the algorithmic, corresponds to Fig. 7e). If the attempt to move the pebbles to the branch w fails (the call to TEST_MOVE returns false), then a different branch vertex is considered. This may happen only in case 2, i.e., when the branch vertex is outside the path $\pi(u, v)$.

Once the pebbles reach the vicinity of w, the algorithm checks if it is possible to create two empty neighbors of w for the swap (lines 9-26). An attempt is made to clear neighbors not occupied by pebbles p, p' (lines 9-13). If no neighbor of w is emptied, it is not possible to swap on w and a different branch has to be found (lines 14-15). If two or more empty neighbors are created, then the swap is possible (lines 25-27), as in Fig. 7b). If only one empty neighbor r exists, then it may be possible to also clear a second one q through r (lines 17-24), as in Fig. 7c). This requires backtracking the pebbles away from w if possible (lines 19-20), moving the pebble from q to r, bringing the p, p' pebbles back to w (line 22) and pushing the pebble at r further away from w if possible (line 23-24), as in Fig. 7c-d). If these actions work, then two empty neighbors of w have been created and it is possible to execute function SWAP_OPS2 as in Fig. 3.

High-level Solver: Given this new way to perform the swap, it is possible to define a new solver in Algorithm 3, which does not have to reduce the PMG challenge into a PPT one. The function tries to continuously move pebbles along their shortest paths to their targets and repeats its main loop until all pebbles have reached them (line 3). The main loop considers the pebbles in an order that is again defined by the target assignment as in the previous solver (line 4). This time, however, it is necessary to first consider a spanning tree of G and give priority to pebbles that are closer to the leaves of the spanning tree. For each pebble p the method first computes the path to its target π (line 6) and then considers the

Algorithm 3: PMG_SOLVER$(G, \mathcal{P}, A^S, A^T)$

```
1  Π ← ∅;
2  A ← A^S;
3  while (A ≠ A^T) do
4      forall the pebbles p ∈ P ordered based on A^T do
5          while A[p] ≠ A^T[p] do
6              π ← COMPUTE_PATH(A[p], A^T[p]);
7              while ∃ pebbles between A[p] and A^T[p] do
8                  p' ← first pebble along π blocking p;
9                  if NEED_SWAP(p, p') or
                   !TEST_CLEAR(G, Π, A[p'], π) then
10                     if SEARCH_SWAP(G, p, p', Π, A)
                       then
11                         break;
12                 return FAILURE;
13 return Π;
```

first pebble p' that blocks this path. At this point the method detects if a swap is necessary between p and p'. This can be achieved with function NEED_SWAP, which returns true if one of the following holds:

- $A^T[p'] \in \pi(A[p], A^T[p])$ (the target of p' is along the path of p) or
- $A[p] \in \pi(A[p'], A^T[p'])$ (the current vertex of p is along the path of p').

These are sufficient conditions for a swap to be executed between p and p' and imply that the two pebbles need to move on opposite directions. If the pebbles do not need to move in conflicting directions, the algorithm tries to clear the path of p by moving p' out of its way with a call to TEST_CLEAR, which tries to move p' towards the closest empty vertex in G while not ending up on any vertex of path π. If the call to TEST_CLEAR fails and it is not possible to evacuate p' from π in this manner, then again a swap operation is needed. If the swap fails, while p and p' need to switch positions so as to make progress, then the problem is not solvable.

Experiments

This section evaluates the proposed algorithms, and compares them against:

- A coupled algorithm, called ODA* with independence detection (ODA*+ID) (Standley and Korf 2011). It is a practical, admissible algorithm for solving multi-agent

	ODA*+ID		PaS		PMG_VIA_PPT_SOLVER		PMG_SOLVER	
Problem	Time (ms)	Path length	Time(ms)	Path length	Time(ms)	Path length	Time(ms)	Path length
Corners	17.65	32	1.94	66(50)	1.49	66(44)	0.52	48(48)
Doubleloop	3.46	25	1.07	31(31)	1.96	143(43)	0.62	39(31)
Small Graph	16.22	25	5.83	145(91)	2.93	127(81)	0.85	100(90)
Stack	∞	n/a	17.82	308(262)	16.91	510(338)	2.20	278(234)
String	1.19	20	0.59	40(32)	0.96	52(44)	0.37	24(24)
Tree	2.3	12	0.9	38(20)	0.79	14(14)	0.47	14(14)
Tunnel	211.74	49	2.03	129(49)	1.39	117(69)	0.95	79(65)
n-2	∞	n/a	37.86	764(560)	1.96	1404(796)	3.385	542(494)

Table 1: Running time (in milliseconds) and solution length for the small benchmarks where ∞ represents a failure to find a solution within 300 seconds. The solution length after smoothing is shown inside the parentheses.

pathfinding problems and can achieve optimality as well as an anytime behavior. This approach modifies state expansion using operator decomposition.

- Push and Swap (PaS) (Luna and Bekris 2011b) is a complete but suboptimal, sequential multi-agent path-finding algorithm. The push primitive evacuates agents along a path of an agent if possible, and the swap primitive exchanges the positions of two adjacent agents while leaving all other agents in place.

To evaluate the proposed approach, a series of challenging instances of multi-robot path planning were considered that have appeared before in the literature. These problems include a set of small benchmarks that are highly coupled so that decoupled planner typically fail on them, as well as larger instances, including maps from an online repository of grid-worlds (Sturtevant 2012). All experiments are performed on an Intel Core i3 3.1GHz machine with 8GB of memory. Results are provided in terms of solution quality and running time.

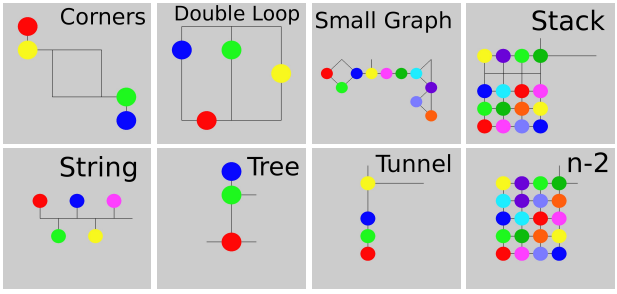

Figure 8: The set of the small benchmarks.

Benchmark Problems: The set of small, benchmark problems are depicted in Fig. 8. Because of the small size of the problems, almost all of the algorithms could return a solution well within a 5 minute time limit. A comparison of the methods can be found in Table 1.

Table 1 shows the running time and the path quality of the corresponding solution. Times of more than 5 minutes are deemed a failure. ODA*+ID computes high quality solutions for small problems, but fails as the complexity of the problem increases. For instance, the algorithm failed for the benchmark Stack with 16 agents. PMG_SOLVER exhibits the best computation time across all benchmarks, while it also performs very well in terms of path quality. It always returns a better solution than PMG_VIA_PPT_SOLVER, while it returns a better solution than PaS in most benchmarks, with the exception of Tunnel after smoothing. Note that for a variation of the small graph benchmark, which it did not have a solution PMG_SOLVER detects that two orders of magnitude faster than PaS.

Large Scale Problems: A second set of experiments with a random grid (500 vertices) and two maps from computer games (2,534 and 10,890 vertices) test scalability. The random grid is populated with 20% random obstacles. Sets of 10 up to 100 pebbles for the grid were tested, while sets of 1 up to 1000 pebbles for the two game maps are placed at random, in mutually exclusive start and target vertices. Average statistics for 20 runs for each setup is depicted in Fig. 9. The top line shows path length, while the bottom line shows the time needed for the algorithms to compute a solution. Dashed lines on the graphs connect points that would otherwise appear significantly higher along the y-axes.

Figure 10: (Left) A randomly generated grid with 500 vertices. (Middle) "Den204d" from Dragon Age: Origins. 2534 total vertices. (Right) "AR0205SR" from Baldur's Gate II. 10890 total vertices (Sturtevant 2012).

ODA*+ID did not manage to find solutions for more than 30 agents in the grid and for more than 20 agents in the two big maps. PMG_VIA_PPT_SOLVER manages to solve all the problems on time, but not as fast as the other two methods as it has to pay an initial cost of reducing the PMG problem into a PPT one. While this is a linear operation, it directly relates to the number of vertices in the graph and has to be paid regardless of the number of agents. As the number of agents increases, the scalability advantages of the PMG_VIA_PPT_SOLVER materialize and it becomes

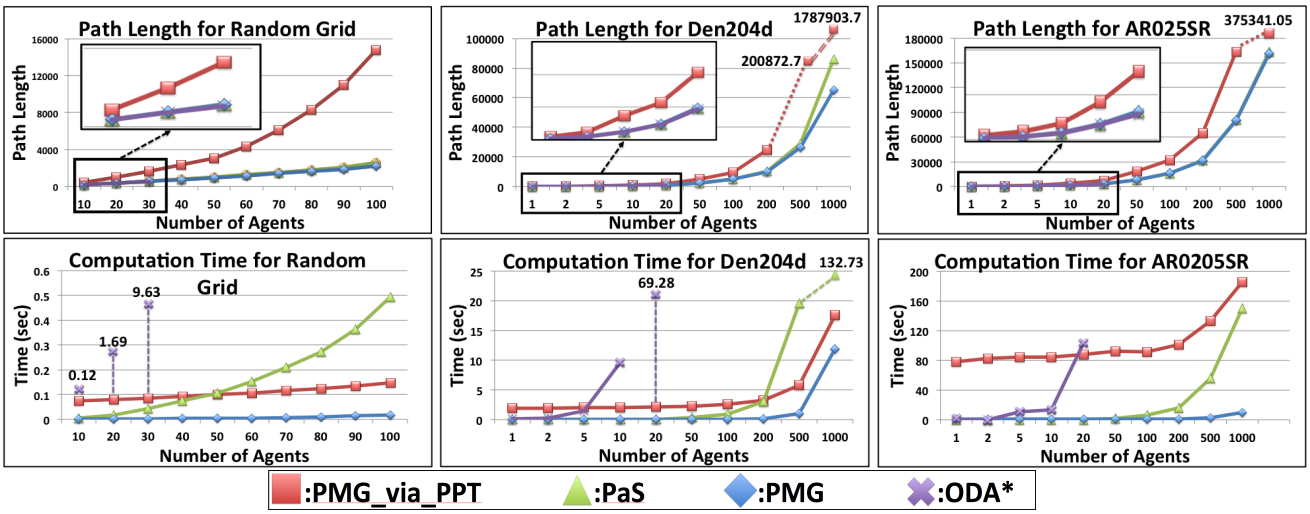

Figure 9: Comparisons between the methods for the length of the solution in the three big environments.

faster than the PaS solution. In terms of path quality the PMG_VIA_PPT_SOLVER gives worse results because of the rotations (as in Fig. 4) that the pebbles will have to follow to move through an M2CC subset of the graph. These big maps correspond to a large M2CC.

On the other hand, the PMG_SOLVER returns a solution to problems typically faster that the alternatives, with the exception of small agent problems in large maps. In most cases, the path length for PMG_SOLVER and PaS is competitive but as the number of agents increases a separation in quality arises in favor of the PMG_SOLVER. For instance, the path length of PMG_SOLVER stays lower than PaS on the "Den204d" map with 2534 vertices as the density of the problem increases and the number of agents reaches 1000. At the same time, PMG_SOLVER returns solutions faster than PaSand manages to maintain the running time within a few milliseconds for the majority of the experiments, with the exception of the denser problems on "Den204d". For these problems the running time for PaS was too big to show on the graph as it needed on average 132 seconds, while the PMG_SOLVER required less than 12.

Discussion

This paper studies how linear-time feasibility algorithms for multi-agent pathfinding can assist in providing better path planners on general graphs. A direct translation of the feasibility algorithms, called PMG_VIA_PPT_SOLVER, provides paths of low quality relative to existing suboptimal planners. At the same time, however, it provided insights on how to improve upon existing solutions and led to the development of a new suboptimal solver, the PMG_SOLVER, which outperforms alternatives in terms of path quality and running time and scales to problems involving thousands of agents, while providing solutions to standard benchmarks where incomplete, decoupled methods fail.

A study on how the PMG_SOLVER framework can provide completeness corresponds to future work. A possible direction for such an analysis corresponds to a sequence of equivalences between algorithms. The completeness of PMG_VIA_PPT_SOLVER is rather straightforward and arises from the feasibility algorithms. Moreover, it is easy to show the equivalence of SEARCH_SWAP with the combination of the FIND_BRANCH and SWAP primitives. More involved is to show that the reduction to a PPT challenge is not needed and that the operation can be performed directly on the graph without any oscillations arising between pairs of agents.

Another issue relates to the asymptotic running time of the algorithms. The work on the feasibility algorithm for trees argues that a solution plan consists of $O(k^2(n-k))$ moves, which in the worst case corresponds to $O(n^3)$ (Auletta et al. 1999). This is also a bound for the running time of the best path planner in the worst case. The running time of PMG_VIA_PPT_SOLVER is dominated by the potentially quadratic number of calls to SWAP, which has a linear worst case cost, as it makes calls to search processes for finding the closest empty vertex so as to push pebbles towards them. Since every other operation of PMG_VIA_PPT_SOLVER is linear time (the FIND_BRANCH function in an aggregate sense), the cost of PMG_VIA_PPT_SOLVER matches that of $O(n^3)$. Showing the equivalence between the two described algorithms will also assist in drawing a running time bound for the PMG_SOLVER algorithm.

The provided results made use only of uninformed search primitives. In many of the problems considered in the experiments, such as grid environments, it is easy to define heuristic values between vertices and use informed search, which will significantly speed up the solution time, especially for the search-based swap primitive of PMG_SOLVER.

It is also interesting to investigate how to potentially utilize the linear-time feasibility tests in order to define optimal solutions by pruning part of the search space during the operation of a coupled planner. Similarly, providing solutions with similar performance to the one achieved by PMG_SOLVER for variations of the basic problem can be useful. This includes situations where the pebbles can move in parallel, or variations which are closer to applications, such as warehouse management, where the targets of the agents change dynamically.

References

Auletta, V.; Monti, A.; Persiano, P.; Parente, M.; and Parente, M. 1999. A Linear Time Algorithm for the Feasibility of Pebble Motion on Trees. *Algorithmica* 23(3):223–245.

Dresner, K., and Stone, P. 2008. A multiagent approach to autonomous intersection management. *Journal of Artificial Intelligence Research* 31:591–656.

Enright, J. J., and Wurman, P. R. 2011. Optimization and coordinated autonomy in mobile fulfillment systems. In *Workshops at the Twenty-Fifth AAAI Conference on Artificial Intelligence*.

Goraly, G., and Hassin, R. 2010. Multi-color Pebble Motion on Graphs. *Algorithmica* 58(3):610–636.

Jansen, M. R., and Sturtevant, N. R. 2008. Direction maps for cooperative pathfinding. In *The Fourth Artificial Intelligence and Interactive Digital Entertainment Conference (AIIDE'08)*.

Khorshid, M. M.; Holte, R. C.; and Sturtevant, N. R. 2011. A polynomial-time algorithm for non-optimal multi-agent pathfinding. In *The Fourth Annual Symposium on Combinatorial Search (SoCS'11)*, 76–83.

Kornhauser, D. M. 1984. Coordinating pebble motion on graphs, the diameter of permutation groups, and applications. Master's thesis, Massachusetts Institute of Technology.

Luna, R., and Bekris, K. E. 2011a. Efficient and complete centralized multi-robot path planning. In *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS-11)*.

Luna, R., and Bekris, K. E. 2011b. Push and Swap: Fast cooperative path-finding with completeness guarantees. In *International Joint Conferences in Artificial Intelligence (IJCAI-11)*, 294–300.

Luna, R.; Oyama, A.; and Bekris, K. E. 2010. Network-guided multi-robot path planning for resource-constrained planetary rovers. In *10th International Symposium on Artificial Intelligence, Robotics and Automation in Space*.

Masehian, E., and Nejad, A. H. 2009. Solvability of multi robot motion planning problems on trees. In *The IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS'09)*, 5936–5941.

Masehian, E., and Nejad, A. H. 2010. A hierarchical decoupled approach for multi robot motion planning on trees. In *The IEEE International Conference on Robotics and Automation (ICRA'10)*, 3604 – 3609.

Peasgood, M.; Clark, C.; and McPhee, J. 2008. A complete and scalable strategy for coordinating multiple robots within roadmaps. *IEEE Transactions on Robotics* 24(2):282–292.

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

Sajid, Q.; Luna, R.; and Bekris, K. E. 2012. Multi-Agent Path Finding with Simultaneous Execution of Single-Agent Primitives. In *Fifth Symposium on Combinatorial Search*.

Sharon, G.; Stern, R.; Goldenberg, M.; and Felner, A. 2011. The increasing cost tree search for optimal multi-agent pathnding. In *The International Joint Conference on Artificial Intelligence (IJCAI'11)*, 662–667.

Silver, D. 2005. Cooperative pathfinding. In *The 1st Conference on Artificial Intelligence and Interactive Digital Entertainment (AIIDE'05)*, 23–28.

Standley, T., and Korf, R. 2011. Complete algorithms for cooperative pathfinding problems. In *Proceedings of the Twenty-Second international joint conference on Artificial Intelligence - Volume Volume One*, IJCAI'11, 668–673. AAAI Press.

Standley, T. 2010. Finding optimal solutions to cooperative pathfinding problems. In *The Twenty-Fourth AAAI Conference on Artificial Intelligence (AAAI'10)*, 173–178.

Sturtevant, N., and Buro, M. 2006. Improving collaborative pathfinding using map abstraction. In *The Second Artificial Intelligence for Interactive Digital Entertainment Conference (AIIDE'06)*, 80–85.

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

Surynek, P. 2009. A novel approach to path planning for multiple robots in bi-connected graphs. In *The IEEE International Conference on Robotics and Automation (ICRA'09)*, 3613–3619.

Wang, K.-H. C., and Botea, A. 2008. Fast and Memory-Efficient Multi-Agent Pathfinding. In *The International Conference on Automated Planning and Scheduling (ICAPS'08)*, 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* 42:55–90.

Wilson, R. M. 1974. Graph puzzles, homotopy, and the alternating group. *Journal of Combinatorial Theory, Series B* 16:86–96.

Yu, J., and LaValle, S. M. 2012. Multi-agent path planning and network flow. In *The Tenth International Workshop on the Algorithmic Foundations of Robotics (WAFR)*.

Yu, J. 2013. A linear time algorithm for the feasibility of pebble motion on graphs. Technical report, http://arxiv.org/abs/1301.2342.