TWO ALGORITHMS FOR THE PACKAGE-EXCHANGE ROBOT-ROUTING PROBLEM

JAMES DRAIN

Abstract. We present and analyze two new algorithms for the package-exchange robot-routing problem (PERR): restriction to individual paths (RIP) and bubbletree. RIP provably produces a makespan that is $O(SIC + k^2)$, where SIC is the sum of the lengths of the individual paths and $k$ is the number of robots. Bubbletree produces a makespan that is $O(n)$, where $n$ is the number of nodes. With optimizations bubbletree can also achieve a makespan of $O((k + l)\log k)$, where $l$ is the longest path from start to goal in the bubbletree subgraph.

1. Statement of the Package-Exchange Robot-Routing Problem

Let the undirected, finite graph $G$ have sets of start and goal nodes $\{s_1, \ldots, s_k\}$ and $\{g_1, \ldots, g_k\}$. Initially there is one phantasmagorical ghost $r_i$ at each $s_i$. In each timestep, the ghosts are allowed to either stay at their current node or move to a neighboring node. No two ghosts can occupy the same node at the same timestep, but they are allowed to pass through each other and swap nodes. The problem is to find a sequence of moves that makes it so that all the ghosts are at their goals at as early a timestep as possible. For a given sequence of moves, that first time when the ghosts are all at their goals is known as the makespan.

2. Applications

There are a few, less ethereal applications. We can replace the $k$ ghosts with $k$ transferable payloads carried by as many robots: packages are allowed to be exchanged between neighboring robots, but no robot can hold two packages simultaneously. Indeed, we can then move the packages along exactly as we would the ghosts, with the understanding that the packages are being carried. We can imagine applications to ride-sharing (or taxis) with passenger transfers (Coltin and Veloso 2014) or package delivery with robots in offices (Veloso et al. 2015), although it is unlikely for the conditions of the problem to be exactly met.

PERR can also be seen as a relaxed form of the multi-agent path-finding problem, MAPF. MAPF is identical to PERR, except that swaps are disallowed. Not all instances of MAPF are solvable; for example MAPF is unsolvable on a graph with only two nodes and two agents that need to change places. In contrast, PERR is always solvable, and both of this paper's algorithms solve all instances of PERR: they are complete. MAPF is a well-studied problem. It was first stated as the pebble-passing problem (Kornhauser 1984), whereas PERR was first posed in 2016 (Hang). The current state of the art optimal solver for MAPF is CBS (Sharon et al. 2013), which can easily be modified to apply to PERR.

Date: written June 26, 2015.
Even with these assumptions and relaxations, PERR (and MAPF) are NP-hard to approximate to within a factor of $4/3 - \epsilon$ (Hang 2016) More specifically, we can encode variants of 3-SAT as instances of PERR (and MAPF) with makespans 3 or 4 depending on whether the corresponding boolean expression is satisfiable. Thus we turn to polynomial algorithms that are asymptotically optimal to within a cruder factor.

The restriction to individual paths (RIP) algorithm is an intuitive algorithm that is empirically near-optimal on graphs with relatively few robots. We also present the bubbletree algorithm, which takes its inspiration from the proof of parallel bubblesort. The basic bubbletree produces provably bounded-suboptimal results for low degree, robot-dense tree graphs with many robots in which the longest individual path length is proportional to the number of robots. More precisely, it gives a makespan that is $O(dn)$, where $n$ is the number of nodes and $d$ is the maximum degree. An optimization produces an $O(n)$ and $O(diameter \times \log k + k)$ makespan for all graphs, where $k$ is the number of robots and diameter is the diameter of the graph. There are however families of graphs that cannot be restricted to a tree while maintaining near-optimal makespans. Incidentally, RIP performs perfectly on the counterexamples we present.

3. Restriction to Individual Paths (RIP) Algorithm

The idea of RIP is to form optimal paths $P_{t,i}$ for each robot at timestep 0 and naively advance them along their paths if possible. Let $P_{t,i}$ denote the shortest individual path for $r_i$ to follow to reach its goal node starting at time $t$. This can easily be accomplished using $A^*$, avoiding the combinatorial explosion of planning for many robots. We can also index into paths. Thus $P_{t,i,0}$ denotes $r_i$’s location at time $t$, and $P_{t,i,1}$ denotes the node $r_i$ wants to go to at timestep $t$. When unambiguous, we abuse notation and conflate robots with their indices.

**Definition 3.1 (advance).** We say that robot $r_a$ advances at time $t$ if $r_a$ moves towards its goal along its shortest path at time $t$ i.e. $P_{t+1,a} \subseteq P_{t,a}$.

If we advance robot $r_i$ at time $t$, then we delete $P_{t,i,0}$ from $P_{t,i}$ to form $P_{t+1,i}$. If there is a cycle of robots $r_1, \ldots, r_m$ such that $r_i$ wants to move to the node occupied by $r_{i+1} \mod m$, then we advance those robots simultaneously in a cycle. If $m = 2$, then this is a happy swap.

**Definition 3.2 (happy swap).** Say that $a$ and $b$ happy swap if they both decrease the lengths of their shortest paths when they swap with each other.

Happy swaps contrast with bully swaps.

**Definition 3.3 (bully swap).** Say robot $a$ bully-swaps $b$ if, when $a$ swaps with $b$, then $a$ shortens its own shortest path, but lengthens $b$’s shortest path.

In RIP, we bully swap $r_a$ past $r_b$ only if $P_{t,b}$ is a subset of $P_{t,a}$. These are known as subset swaps.

**Definition 3.4 (subset-swap).** Say that $a$ subset-swaps $b$ if $P_{b,t} \subset P_{a,t}$ and $a$ swaps with $b$.

Further define directed swap.

**Definition 3.5 ((directed) swap need $(a,b)$).** Say that $a$ swap-needs $b$ if, after removing all other robots, it is impossible to advance $a$ and $b$ to their goals without $a$ bully-swapping $b.$
The advantage of subset swaps $a \leftrightarrow b$ is that they are guaranteed to resolve swap needs $(a, b)$ when $P_{t,b}$ is a parallel subset of $P_{t,a}$ i.e. the nodes in $P_{t,b}$ appear in the same order in $P_{t,a}$. In this case we write $P_{t,b} < P_{t,a}$. If robot $r_j$ is subset-swapped at time $t$ by robot $r_i$, then we prepend $P_{t,i,0}$ to $P_{t,j}$ to form the shortest path $P_{t+1,j}$.

In the pseudocode for RIP, the inner while loop advances robots that can move without cycling or swapping. We then perform subset swaps. Finally we resolve any cycles of unmoved robots. (RIP is complete for any ordering of these three operations.)

In general, robots move simultaneously in a given timestep, but we have to plan serially which robots to move. For example, it is possible that unmoved robots. (RIP is complete for any ordering of these three operations.)

We first need to verify that, if not all robots are at their goal, then at least one robot moves. Define a robot $r_i$’s bottleneck set $B_i$ at time $t$ to be the set of robots that want to advance to the same node as $r_i$. Define $f : [k] \rightarrow [k] \cup V$ by letting $f_i(i)$ denote the (index of) the robot occupying $P_{t,i,1}$ if there is one, and otherwise letting it denote $P_{t,i,1}$ itself. At the beginning of timestep $t$, start at any robot $r_a$ that is not at its goal, and follow the chain $r_a, f_t(a), f_t^{(2)}(a), \ldots, f_t^{(m)}(a)$ until first reaching an unoccupied node in the graph or else repeating a robot. If this chain terminates with $f_t^{(m)}(a) \in V$, then either the robot $f_t^{(m-1)}(a)$ will claim that vertex, or else some other robot in its bottleneck set will claim that vertex. If the chain terminates with $f_t^{(m)}(a)$ equaling an earlier $f_t^{(m-i)}(a)$ with $i > 1$, then either some subset of $f_t^{(m-i)}(a), \ldots, f_t^{(1)}(a)$ will move before detecting cycles, or else they will all move in a cycle. Finally, if $f_t^{(m)}(a) = f_t^{(m-1)}(a)$, then some robot in $f_t^{(m-2)}(a)$’s bottleneck set $B$ will move: some robot in $B$ will swap past $f_t^{(m)}(a)$, or else every robot in $B$ will be swapped.

Now that we know some robot moves every timestep, let us define the notion of swap risk.

**Definition 4.1** (directed) swap risk. Say there is a (directed) swap risk $(a, b)$ if there is already a swap need $(a, b)$ or if, after removing all other robots, it is possible to introduce a swap need $(a, b)$ by advancing $a$ and $b$ along their respective shortest paths.

The proof of completeness works by defining the potential function $\Phi = \sum_{i=1}^k d(r_i, g_i) + \sum_{i=1}^k \sum_{j=1}^k s(i, j)$, where $s(i, j) = 1$ if $i$ swap-risks $j$, and otherwise $s(i, j) = 0$ (we always have $s(i, i) = 0$.) Observe that $\Phi \leq k^2 + SIC$. It is sufficient to show that a single action decreases $\Phi$ by at least 1; performing $m$ actions simultaneously will thus decrease the potential by at least $m$.

By the definition of swap risk, neither advancing a robot along its unimpeded path nor resolving a cycle of robots introduces a swap risk. Advancing an unimpeded robot and
Algorithm 1: RIP

1 $t \leftarrow 0$;
2 for $i=1,\ldots,k$ do
3 \hspace{1em} $P_{0,i} = \text{shortestPath}(s_i,g_i)$;
4 \hspace{1em} for $i=1,\ldots,k$ do
5 \hspace{2em} $Q_{t,i} \leftarrow P_{t,i,0}$;
6 while some agent is not at its goal do
7 \hspace{1em} mark all robots as unmoved;
8 \hspace{1em} movedAnyone = true;
9 \hspace{1em} while movedAnyone do
10 \hspace{2em} movedAnyone $\leftarrow$ false;
11 \hspace{2em} for $i = 1,\ldots,k$ do
12 \hspace{3em} if $P_{t,i,1}$ is unoccupied then
13 \hspace{3em} mark $r_i$ as moved;
14 \hspace{3em} movedAnyone $\leftarrow$ true;
15 \hspace{3em} delete $P_{t,i,0}$;
16 \hspace{1em} for $i = 1,\ldots,k$ do
17 \hspace{2em} if $r_i$ is unmoved then
18 \hspace{3em} if there is an unmoved robot $r_j$ occupying $P_{t,i,1}$ then
19 \hspace{4em} if $P_{t,j} \prec P_{t,i}$ then
20 \hspace{5em} mark $r_i$ and $r_j$ as moved;
21 \hspace{5em} prepend $P_{t,i,0}$ to $P_{t,j}$ to form $P_{t+1,j}$;
22 \hspace{5em} delete the zeroth entry of $P_{t,i}$ to form $P_{t+1,i}$;
23 \hspace{3em} resolve any cycles of unmoved robots by moving all involved robots in a circle, updating the $P$'s appropriately and marking those robots as moved;
24 \hspace{1em} for $i = 1,\ldots,k$ do
25 \hspace{2em} $P_{t+1,i} \leftarrow P_{t,i}$;
26 \hspace{2em} $Q_{t+1,i} \leftarrow P_{t+1,i,0}$;
27 $t \leftarrow t + 1$;
28 Return $Q$;

resolving a cycle decrease the sum of distances. Thus both actions decrease the potential and we only have to consider bully swaps, which we know do not increase the sum of distances.

Before continuing, we need to introduce the notion of parallel paths.

Definition 4.2 (parallel paths). Let $P_{t,c} \cap P_{t,d} = \{a_0,\ldots,a_n\}$ be nonempty and let those nodes appear in the same order $a_0,\ldots,a_n$ in both $P_{t,c}$ and $P_{t,d}$. Then $P_{t,c}$ and $P_{t,d}$ are said to be parallel paths and we write $P_{t,c} \Rightarrow P_{t,d}$ (or, equivalently, $P_{t,d} \Rightarrow P_{t,c}$). Conversely, if $n \geq 1$ and the nodes appear in that order in $P_{t,c}$ but in opposite order in $P_{t,d}$, then we say the two paths are antiparallel and write $P_{t,c} \Leftarrow P_{t,d}$ (or, equivalently, $P_{t,d} \Leftarrow P_{t,c}$). If $n = 1$, the paths are more specifically trivially parallel.

Lemma 4.3. Shortest paths are either parallel or antiparallel (or nonintersecting).
Proof. Let $P_{t,c}$ and $P_{t,d}$ be two shortest paths and let $\{x, y, z\}$ appear in both paths, and let $x, y, z$ appear in that order in $P_{t,c}$. It is sufficient to show that $\{x, y, z\}$ must appear in the same or opposite order in $P_{t,d}$; then we could inductively show that the whole paths are (anti-)parallel by showing that the first three common nodes of $P_{t,c}$ have (anti-)parallel counterparts, then common nodes two through four must also have (anti-)parallel counterparts, then common nodes three through five etc. Thus, for the sake of contradiction and wlog, assume that $x$ comes between $y$ and $z$ in $P_{t,d}$. By the shortest paths hypothesis applied to $P_{t,d}$, we have $d(x, z) < d(y, z)$. But applying the shortest paths hypothesis to $P_{t,c}$ gives $d(y, z) < d(x, z)$, contradiction.  

Note that if two shortest paths are antiparallel and we extend one of them to another shortest path, then they will still be antiparallel. Conversely, if two paths are parallel, then we can remove extremal nodes while maintaining parallelism. However, neither parallelism nor antiparallelism is transitive in general. For example, let $P_1 = a, b, c$, $P_2 = a, b, d$, and $P_3 = d, b, c, e$. Then $P_1 \Rightarrow P_2$ and $P_1 \Rightarrow P_3$, but $P_2 \not\Rightarrow P_3$.

Parallel paths give a better sense of what subset swapping does in RIP. If $P_a \Rightarrow P_b$, then the only way $a$ can subset-swaps $b$ is if $P_a$ has only two nodes: the node $v_b$ occupied by $b$, and the node $g_b$ occupied by $a$. In this case, $a$ immediately brings $b$ to its goal. If $P_b \Rightarrow P_a$ and $a$ subset-swaps $b$, then this is a bully swap as $P_{t+1,b} \subset P_{t,a}$ is a shortest path, and longer than $P_{t,b}$.

Let $a$ bully swap $b$ and let $x \neq a, b$. When a subset-swaps $b$, they resolve the swap risk $(a, b)$. This is proved after formally characterizing swap risks in proposition 4.8.

Clearly no swap risk $(x, y)$ is introduced for $y \neq a, b$. Since $a$ advances, there is no new swap risk between $x$ and $a$, either. Since $g_a$ is not on $P_{t+1,b}$, there is also no new swap risk $(b, a)$. (We will prove this after characterizing swap risks.) It is thus sufficient to prove the following lemma:

**Lemma 4.4 (Main Monovariant Lemma).** The only way bully subset-swapping a past $b$ can incur a new swap risk between $x$ and $b$ is if it is a directed swap risk $(b, x)$, and if swapping this way simultaneously resolves a swap risk $(a, x)$.

The proof requires a couple lemmas and a definition.

**Lemma 4.5.** RIP maintains the shortest paths invariant. That is, every $P_{t,a}$ is at all times a shortest path from $P_{t,a,0}$ to $g_a$.

Proof. Assume that all paths are shortest paths at time $t$. Clearly advancing a robot along its path keeps it on a shortest path. Let $a$ subset-swaps $b$. Let $P_{t,b} = v_0, v_1, \ldots, v_n$, (with those vertices appearing in that order). If $n = 1$, then it is possible that the two paths are antiparallel, in which case the swap is a happy swap and the robots advance, maintaining the shortest paths invariant. This is the only case in which the paths are antiparallel: if they are antiparallel, then we need $v_n$ to appear in $P_{t,a}$ before $v_0, v_1, \ldots, v_n$. But we know that $v_0 = P_{t,b,0} = P_{t,a,1}$. Thus $P_{t,a}$ begins $v_n, v_0$, hence $n = 1$. Now consider when the two paths are parallel. If $b$ is at its goal before being swapped, then its path after being swapped is length one, which is best possible. If $b$ is not at its goal, then $P_{t+1,b} = P_{t,a,0}, v_0, v_1, \ldots, v_n$ with $n > 1$. Since $P_{t,a,0}, v_0, v_1, \ldots, v_n$ is a shortest (sub)path from $P_{t,a,0}$ to $v_n$, it follows that $P_{t+1,b}$ is indeed a shortest path.  

In the following lemmas, all paths are shortest paths, as appear in RIP.
Lemma 4.6. If \( P_c \Rightarrow P_d \) are parallel shortest paths, then there is at most one of the swap risks \((c, d)\) and \((d, c)\), i.e. parallel swap risks are inherently directed. If \( P_c \Leftrightarrow P_d \) are antiparallel shortest paths, then we can have any combination of swap risks \((c, d)\) and \((d, c)\).

Proof. Let \( P_c \Rightarrow P_d \). The proof uses the fact that we cannot simultaneously have \( g_d \in P_c \) and \( g_c \in P_d \); otherwise we would have \( g_c, g_d \) appear in that order in \( P_d \), and in reverse order in \( P_c \). For two robots on parallel paths to incur a swap need, it must inevitably get to the point where one of the robots, say \( c \), is not yet at its goal, but its next node is occupied by \( d \), who is at its goal. By the fact we just mentioned, that means we have at most one of \( s(c, d) \) and \( s(d, c) \). Now let \( P_c \Leftrightarrow P_d \). If \( P_c = v_0, v_1 \) and \( P_d = v_1, v_0 \), then we have neither \( s(c, d) \) nor \( s(d, c) \). If \( P_c = v_0, v_1, v_2 \) and \( P_d = v_3, v_2 \), then we can incur a swap need by advancing \( P_d \) one step to \( v_2 \). If \( P_c = v_0, v_1, v_2, v_3 \) and \( P_d = v_4, v_3, v_2, v_1 \), then we can incur a swap need \((c, d)\) by advancing \( d \) to its goal, or a swap need \((d, c)\) by advancing \( c \) to its goal.

We have the following characterization of swap-needs:

Proposition 4.7. Let \( P_c \) and \( P_d \) be shortest paths. Then \( c \) swap-needs \( d \) if and only if \( d \)'s goal is on \( c \)'s path, \( c \) and \( d \)'s paths are parallel, and \( d \)'s path is a subset of \( c \)'s i.e. \( P_d \) is a subsequence of \( P_c \). In notation, \( c \) swap-needs \( d \) if and only if \( g_d \in P_{t,c} \), \( P_{t,c} \Rightarrow P_{t,d} \), and \( P_{t,d} \subseteq P_{t,c} \).

Proof. If those hypotheses hold, then \( c \) cannot get to any node on \( P_d \) before \( d \). In particular, \( d \) gets to \( g_d \) before \( c \). To show the hypotheses are necessary, consider negating each one. We need \( g_d \) on \( P_c \) to have a swap risk \((c, d)\). Indeed, the only way \( c \) can be stuck before reaching its goal is if its next node is currently occupied by \( d \), and \( d \) does not want to move – because it is already at \( g_d \). If the paths are not parallel, then they are either nonintersecting or else antiparallel. Clearly there is no swap need if the paths are nonintersecting. If they are antiparallel, let their intersection be \( v_0, \ldots, v_m \), appearing in that ordering in \( P_c \) with \( m > 0 \). We can get both robots to goal by advancing (or keeping) \( c \) to \( v_0 \) and advancing (or keeping) \( d \) to \( v_1 \). We can then either immediately swap the two robots, and then send them along their nonintersecting remaining paths. Or else we can move one of the robots off its \( v_i \), send the other robot to goal, and then send the first robot to goal. Finally, assume the paths are parallel but there is some node \( v \) on \( P_d \) that is not on \( P_c \). Then \( d \) must go through \( v \) before stymying \( c \) at \( g_d \). But then there is no swap need, as \( c \) can advance to \( g_c \) while \( d \) waits at \( v \). \( \square \)

This then allows for a characterization of swap risks. Let \( P_a = a_0, \ldots, a_n \) with \( g_b = a_m \) and \( P_b \) be shortest paths.

Proposition 4.8. Robot \( a \) swap-risks \( b \) if and only if either
1. \( P_a \Rightarrow P_b \), \( g_b \) is on \( P_a \), and \( a \) is not currently occupying \( g_b \) i.e. \( a_0 \neq g_b \).
2. \( P_a \Rightarrow P_b \), \( g_b \) is on \( P_a \), and \( P_{a,0}, \ldots, P_{a,m-1} \) is not a subset of \( P_b \). Equivalently, \( P_{a,0}, \ldots, P_{a,m} \) is not a subset of \( P_b \).

Proof. If (1) holds, then we cannot have \( P_{a,0} \) on \( P_b \), because it must come after \( g_b \) by antiparallelism. Thus we can incur a swap need \((a, b)\) by advancing \( b \) directly to \( g_b \). Conversely, if \( P_a \Rightarrow P_b \), then to incur a swap need \((a, b)\), we need that \( g_b \) can be occupied by \( b \) before \( a \): otherwise, if \( b \) blocked \( a \)'s progress, we would always be able to move \( b \) or happy swap \( a \) and \( b \).
\[ P_{a,0}, \ldots, P_{a,m-1} \] not being a subset of \( P_b \) captures the condition that \( b \) can get to \( g_b \) before \( a \). One way of thinking of this target condition is that, if we truncate \( a \)'s path to \( P_{a,0}, \ldots, P_{a,m-1} \), then we do not have that \( b \) swap-needs \( a \). Thus, if (2) holds, then there is some way to finalize \( b \) to \( g_b \) before \( a \), again incurring a swap need. Conversely, we need \( g_b \) on \( P_a \) to have a swap risk \((a, b)\), and some way for \( b \) to get to \( g_a \) before \( a \) without swapping. \[ \square \]

It might seem mysterious that there can be swap risks between robots on antiparallel paths. The explanation is that those paths can become trivially parallel. There is a swap need if and only if that trivial intersection is a goal node \( g_a \) occupied by its goalie \( a \).

It follows from the characterization of swap risk that subset-swapping \( a \) past \( b \) at time \( t \) resolves the swap risk \((a, b)\), as the two new paths are parallel and \( P_{t+1,a,0}, \ldots, P_{t+1,b} \) are shortest paths, and \( b = P_{t+1,b,1}, \ldots, g_b \subset P_{t+1,b} \). Also, it does not introduce a swap risk \((b, a)\), as \( g_a \) is not on \( P_{t+1,b} \). The paths are parallel before and after swapping. After swapping, \( P_{t+1,a,0}, \ldots, P_{a,\text{index}(g_b)-1} \) is not a subset of \( P_{t+1,b} \), so there is no longer a swap risk \((a, b)\).

**Lemma 4.9.** If \( P_c \Rightarrow P_d \) are shortest paths, then we can have at most one of \( g_d \in P_c \) and \( g_c \in P_d \).

**Proof.** Otherwise we would have \( g_d, g_c \) appear in that order in \( P_c \) and \( g_c, g_d \) appear in that order in \( P_d \), and the paths would be antiparallel, contradiction. \[ \square \]

We can now return to the proof of the main monovariant lemma.

**Proof of Main Monovariant Lemma.** Denote the vertex that \( b \) is pulled from and that a swaps to as \( v_b \). Similarly, denote the vertex that \( a \) leaves and \( b \) is pulled to as \( v_a \). The only way pulling \( b \) back can introduce a swap need between \( x \) and \( b \) is if \( x \) can advance along its path to \( v_b \), and in doing so maintain the new swap risk. Denote \( x \)'s shortest path from \( v_b \) as \( P_{t,x} \), and denote \( a \) and \( b \)'s shortest paths after swapping as \( P_{t,a} = P_{t,a} \setminus P_{t,a,0} \) and \( P_{t,b} = P_{t,b} \setminus P_{t,a,0} \). We will work through three cases. Cases (1) and (3b) cannot give rise to a new swap risk between \( x \) and \( b \). Cases (2) and (3a) allow for a new swap risk \((b, x)\), but only by resolving the previous swap risk \((a, x)\).

1. (1) Assume \( P_{t,x} \subsetneq P_{t,b} \). Since \( P_{t,x,0} = P_{t,b,1} \), then we must have that \( P_{t,x} \) routes through \( v_a \), since otherwise they would only intersect in \( v_b \) and be parallel. There is no swap risk then: if all other robots are removed, then \( b \) and \( x \) are either immediately forced to happy swap, or else \( x \) takes a detour during which \( b \) moves to \( v_b \), and then their remaining paths have empty intersection.
2. (2) Assume \( P_{t,x} \supsetneq P_{t,b} \) and that they only have \( v_b \) in common. For there to be a swap risk, we need \( v_b = g_x \); otherwise we would resolve the swap risk as soon as we move \( x \). So let \( v_b = g_x \). That means that there is a swap risk \((a, x)\) at time \( t \), as \( x \) could advance to \( v_b = g_x \) and make \( a \) swap-need it. Conversely, advancing \( a \) to \( v_b \) resolves this swap risk.
3. (3) Assume \( P_{t,x} \supsetneq P_{t,b} \) and that they have at least two vertices in common, one of which we know must be \( v_b = P_{t,x,0} = P_{t,b,1} \). There are two subcases: \( g_x \in P_{t,b} \), and \( g_b \in P_{t,x} \), which correspond to swap risks \((b, x)\) and \((x, b)\) respectively. We use lemma 4.10 to further break into the following two subcases
   (a) \( g_x \in P_{t,b} \). Then \( v_a \notin P_{t,x} \), since, if so, we would have \( v_b, v_a \) in that order in \( P_{t,x} \), and \( v_a, v_b \) in that order in \( P_{t,b} \). Thus, while \( b \) sits at \( v_a \), \( x \) can advance from \( v_b \) to \( g_x \) and incur the swap-need \((b, x)\). This means that there was a swap

---

\[ 7 \]
risk \((a,x)\) at time \(t\), as \(x\) could similarly have advanced to \(v_b\) and then \(g_x\) while \(a\) sat at \(v_a\). Furthermore, we assumed that there was no swap risk \((b,x)\) at time \(t\). That is, regardless of how we were to advance \(b\) and \(x\), \(b\) would make it past \(g_x\) and eventually to \(g_b\). This same reasoning shows that there is no swap risk \((a,x)\) at time \(t'\): by parallelism, \(g_x\) is the last node from \(P_{t',x}\) to appear on \(P_{t',a}\); if we restrict to advancing \(x\) and \(a\) at time \(t'\), then we are forced to advance \(a\) past \(g_x\), and eventually to \(g_a\).

\(b\) \(g_b \in P_{t',x}\) and there is a swap risk \((x,b)\) at time \(t'\) but not \(t\). Let \(P_{t,x} = x_0, \ldots, x_n\), and let \(x_m = g_b\). By the characterization of swap risk, for there to be no swap risk at time \(t\) we need \(x_0, \ldots, x_{m-1}\) to not be a subset of \(P_{t,b}\). But then \(x_0, \ldots, x_{m-1}\) is not a subset of \(P_{t',b}\) either: that would only be possible if \(x_0 = v_a\), which is not the case. Thus there cannot be a new swap risk \((x,b)\) at time \(t'\).

\[\square\]

We can finally prove the theorem.

**Theorem 4.10.** RIP terminates in \(O(k^2 + SIC)\) timesteps with every robot at its goal.

**Proof.** By the monovariant lemma, the potential function \(\Phi \leq k^2 + SIC\) decreases by at least 1 every timestep until the robots meet their goals. \(\square\)

### 5. Bubbletree

#### 5.1. Comparison with Parallel Sorting

In addition to the relation to MAPF, there is a similarity between PERR and parallel bubble sorting, especially for dense maps. For the case of linear maps, we can think of the robots as being numbered by the order of their goals, and think of swapping them as swapping two array entries in bubblesort. Of course, a substantial advantage of PERR over sorting with neighbor-swapping operations is that we already know the destination of each entry. Consider the case of a binary tree. If we choose the root node in the “middle” of the tree such that the “left” subtrees and “right” subtrees form an equipartition of the root node’s children, then, in \(n\) steps can we move all robots with left destinations to the left subtrees and vice versa. We can then recursively call this algorithm on the two subtrees in parallel, giving an \(O(n)\) makespan, which is best possible. The bubbletree algorithm generalizes to higher degree trees. It works by forming target subtrees for each subtree \(T_c\) defined by a child \(c\) of the mid node. In the pre-recursive phase, we can again send all robots to their target subtree in \(O(n)\) steps.

A disadvantage of PERR over sorting on trees with neighbor-swapping operations is that the graph topology for PERR is specified in advance. We have some freedom in choosing the spanning tree (or forest), but there is much more freedom in building a balanced heap, say, in which the entries need only be partially ordered and in which new nodes can be spontaneously produced.

Bubble sorting is a rightly maligned sorting algorithm: its quadratic runtime is much worse than the \(n \log n\) bound on comparison sorting achieved by quicksort and mergesort, and, in practice, bubble sort is significantly slower than other quadratic sorting algorithms like insertion sort and selection sort. Parallel bubble sorting achieves the maximum possible speed up, proportional to the number of processors if there are at least as many entries as processors (Silva 2016). Even an \(O(n)\) runtime is disappointing for parallel sorting, however. There is a known algorithm, which uses the idea of expander graphs, that attains the optimal
\(O(\log n)\) runtime, although the big-\(O\) hides prohibitive constants (Ajtai et al. 1984). And there are practical parallel sorting algorithms like odd-even sort that run in \(\log^2 n\) time (Haberman 1972). Nevertheless, since robots have to physically move one step at a time, we cannot expect \(n\) robots to reach their goals in under \(n\) steps. Thus parallel bubblesorting is a natural place to look for inspiration and the hope of optimal big-\(O\) results. As mentioned above, the analogy between PERR and parallel bubble sorting is still imperfect however, for four reasons:

1. We already know the destination of each robot. For example, if we were allowed to perform insertions, we could sort the robots in \(n\) steps. This suggests that we might expect to get even better performance in PERR than that obtained by bubblesort-esque comparison sorts.

2. PERR maps can have very few robots compared to the total number of nodes.

3. Parallel bubble sorting is only designed for linear arrays, whereas PERR allows for arbitrary graph topologies.

4. We care much more about the makespan than the runtime. That is, we do not mind as much about excess comparisons as long as the robots do not have to perform them in the real world.

The second and third differences will require generalizing parallel bubblesort. The first difference allows us to use specialized heuristics that do not transfer to bubblesorting on general graph topologies. Empirically, RIP is outperformed by bubblesort and shearsort, a cousin of bubblesort specialized for two-dimensional arrays. This is not very surprising given how general-purpose RIP is. We have not yet implemented bubbletree.

6. BUBBLETREE OVERVIEW

The idea of bubbletree is to restrict the graph to a tree \(T \subset G\) that contains all the start and goal vertices. We can also form a forest, as long as each pair \(\{s_i, g_i\}\) of start and goal vertices is contained in the same tree. Throughout, we will give the description of the algorithm for a single tree. Suggestions for how to form the forest are given in the optimizations section.

We want to have some guarantee that the robots are making progress. To this end, designate a mid node \(v_m\) such that each of the subtrees defined by its children has at most \(k/2\) goal vertices: the notion of progress is relative to \(v_m\). (One way to find such a mid node is to perform a breadth-first search from an arbitrary vertex \(v\), and mark every vertex \(v'\) with the number of vertices in the subtree it defines. For example, every leaf \(l\)'s subtree \(T_l\) is size 1. If \(|T_c| \leq |T|/2\) for every child \(c\) of \(v\), then we are done. Otherwise let \(d\) be the unique node for which \(|T_d| > |T|/2\). Then \(d\) is a satisfactory mid node.)

Every robot has an initial subtree: the \(T_c\) for the closest child \(c\) of \(v_m\) (the one irrelevant exception is that a robot \(r_i\) with \(s_i = v_m\) has no initial subtree.) Every robot \(r_i\) also has a target subtree: the \(T_c\) for the closest child \(c\) of \(v\) closest to \(g_i\) (there is a special subroutine for handling the robot \(r_m\) with \(g_m = v_m\). If such a robot exists, then it has no target subtree.) We form groups \(G^I_c\): those robots with a common initial subtree \(T_c\), and groups \(G^T_c\): those robots with a common target subtree \(T_c\). All robots that are not currently at their target tree are called migrants. We let \(M_c \subset G^I_c\) denote the set of robots with initial tree \(T_c\) that have not yet made it to their goal tree. The algorithm is designed so that each migrant set \(M_c\) weakly shrinks with each timestep. Once every migrant robot has made it to its target
we simultaneously move such a chain, it must be because we form such a chain and move them all simultaneously deeper into T. Do not move T into the v

to v

v

v

{T

{c

v

{M

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v

v
$d_{\text{deg}}d_{\text{depth}} + 4k) + f(d, n/2) \leq (d_{\text{deg}}n + 4n) + f(d_{\text{deg}}, n/2)$. It can easily be verified that $g(d, n/2) = 2dn + 8n$ satisfies the recurrence $g(d, n) = (dn + 4n) + g(d, n/2)$. Thus $f(d_{\text{deg}}, n) \leq 2d_{\text{deg}}n + 8n$.

Of course, this result is not very impressive if there are not very many robots, or if the max-degree is unbounded. In the optimizations section we present two modifications that give a provable makespan of $O(\text{diam} \times \log k + k)$, where diam is the diameter of the bubbletree. They are

1. Choosing $v_m$ so that its subtrees are goal-balanced, rather than node-balanced.
2. Moving as many $M_c$ as possible in a given timestep, rather than just the focal $M_c$.

We would attain an appealing linear result without the $\log k$ factors. Unfortunately, no algorithm that prunes down to an acyclic graph can achieve an $O(l + k)$ makespan in all cases, where $l$ is the max distance from start to goal in the entire graph. Consider $n/\log n$ robots on a circular graph with $(s_i, g_i) = (i\log n, (i + 1)\log n + 1) \mod n)$. The union of the shortest paths is the entire graph. Thus we must destroy some robot’s shortest path if we subtract an edge. The optimal makespan is $\log n + 1$, obtained by advancing all the robots simultaneously in the circle. However, the optimal makespan with an edge deleted is at least $n - (\log n + 1) = \Omega(n) = \omega(k + 1)$.

There may still be hope of a linear result in terms of diam or even $l$ for a tree graph ($l$ denotes the longest distance between some $s_i$ and $g_i$.) For example, if we could guarantee that the subtrees we recursively call bubbletree on had either diameters or longest paths that shrank by a constant factor, then we could prove a linear result (assuming this lengthens the pre-recursive phase by only a constant factor).

However, we cannot guarantee shrinking diameter if one of the subtrees is linear. We cannot even guarantee constantly shrinking diameter after many steps, as the tree could have a linear backbone with highly-branched offshoots, say with $2^{i-1}$ nodes at distance 1 from the $(2^i)^{th}$ node of the backbone (or at distance $\log(2^i - 1)$, if the tree has bounded degree). We cannot guarantee constantly shrinking maximum path length, either. For example, we could again have a tree with a long backbone and exponentially shrinking offshoots. Say there is an offshoot of $2^{\lg k - i}$ goal nodes emanating from the $i^{th}$ node of the backbone, and say the backbone itself contains all the start nodes and is very long. If the tail of the backbone is comprised of start nodes from the $\lg k$ different offshoots, then there will be a recursive call at all depths 1, ..., $\lg k$ for which the maximum length to goal will decrease by at most $\lg k$. If the backbone is sufficiently long, it will also be the case the diam $\times \lg k$ dominates $k$.

The optimization of advancing infiltrators (which does not wait for recursive calls in order to advance the robots starting at the tail of the backbone) effectively reduces diam to $\lg k$ in time for the first recursive call in this example. But we do not know if it is strong enough to give a linear makespan in general.

7. Basic Bubbletree pseudocode

8. Experimental Results

We tested RIP on: the Dragon Age Origins map brc2d (Sturtevant 2012) with up to 50 robots; on 20x15 grid maps with randomly generated obstacles, and under a quarter as many robots as free nodes; and on linear and square arrays with as many robots as nodes. For both the DAO and 20x15 grid maps, we used the exact same (randomly generated) start
Algorithm 2: Basic Bubbletree algorithm High-Level

1 \( T \leftarrow \text{formBubbleTree}(G); \)
2 \( v_m \leftarrow \text{findMidVertex}(T); \)
3 \textbf{for each child } \( c \) \textbf{of } \( v_m \) \textbf{do}
4 \textbf{form the migrant set } \( M_c \) \textbf{of robots in } \( T_c \) \textbf{that are not in their target tree;}
5 \textbf{for each child } \( c \) \textbf{of } \( v_m \) \textbf{do}
6 \textbf{assign a (random) ordering to the robots in } \( M_c; \)
7 \textbf{while not all } \( M_c \) \textbf{are empty do}
8 \textbf{if } \( v_m \) \textbf{is occupied, by say } \( r \in G_T \) \textbf{then}
9 \textbf{if } \( \text{focalM} \) \textbf{is nonempty then}
10 \( \text{focalM} \leftarrow M_c; \)
11 \( \text{priorityMove}(M_c); \)
12 \textbf{else}
13 \( \text{moveChain}(G^T_{c}); \)
14 \textbf{else}
15 \( \text{focalM} \leftarrow \text{randomNonemptyM}(); \)
16 \( \text{priorityMove}(M_c); \)
17 \( \text{moveRmtoVm}(); \)
18 \textbf{for each depth-1 subtree } \( T_c \) \textbf{in parallel do}
19 \( \text{bubbleTree}(T_c); \)

and goal positions as in (Ma 2016). We compare our results against the optimal flow-based and cbs-based solvers used in (Ma 2016). We compare the linear and square array results against parallel bubblesort and shearsort, which are specialized for sorting linear and 2D arrays, respectively.

8.1. DAO Map. Adaptive CBS is an optimal solver, but somewhat slower than the flow-based solver introduced in (Ma 2016). Table 1 thus demonstrates that RIP performs near optimally on the DAO tests. Note that the “super-optimal” RIP makespans are due to the fact that many of the instances that Adaptive CBS did not solve in the allotted time have smaller than average makespans. In our tests, the DAO map is robot-sparse: it has 254,930 cells, and at most 50 robots. These tests corroborate the intuition that RIP performs well on maps with relatively few conflicts.

The runtime scaling for RIP appears to be linear with the number of robots, which also makes sense given the relative paucity of collisions: it is the case that almost every step in the inner while loop should advance a robot. For contrast, the runtime scaling for Adaptive CBS is more dramatic – 63% of all tests with 50 robots did not even finish in the allotted five minutes. For Adaptive CBS, the median runtime is over three times that of the average completed runtime, suggesting the distribution of runtimes for a map with randomly seeded starts and goals is heavily-tailed as well.

8.2. 20x15 Grids. Here density denotes the fraction of nodes that were replaced with impassable obstacles. The flow-based solvers are presented in (Ma 2016). The suboptimal flow
| Adaptive CBS | RIP |
|-------------|-----|
| Robots | Fraction Solved | Makespan | Time | Makespan | Runtime |
| 5 | 1.00 | 732.1 | 0.34s | 732.13 | 1.23s |
| 10 | 1.00 | 809.03 | 17.75s | 809.17 | 1.40s |
| 15 | 0.97 | 882.28 | 3.51s | 880.90 | 2.1s |
| 20 | 0.87 | 905.15 | 5.43s | 908.03 | 2.63s |
| 25 | 0.97 | 931.34 | 22.73s | 937.40 | 2.8s |
| 30 | 0.87 | 942.19 | 29.03s | 938.10 | 3.77s |
| 35 | 0.77 | 963.13 | 50.80s | 969.33 | 4.67s |
| 40 | 0.53 | 974.25 | 30.50s | 965.80 | 5.67s |
| 45 | 0.70 | 974.1 | 77.49s | 962.87 | 5.77s |
| 50 | 0.37 | 943.36 | 86.76s | 962.43 | 6.43s |

The makespans from RIP are competitive with the optimal makespans on the easier problems, and the RIP makespan is never more than 30% worse than the optimal makespan. Importantly, the runtime for RIP appears linear with the number of robots and invariant of the number of obstacles, whereas the flow-based solvers demonstrate their non-polynomial nature on the hardest instances.

With the tremendous focus on swaps in the proof for RIP, we might hope that RIP performs very few swaps. However, RIP makes more swaps than the optimal flow algorithm in all instances, even though the flow solution does not try to minimize the number of swaps. RIP performs about as many swaps as suboptimal flow: it performs slightly fewer swaps in the instances with fewer robots, and slightly more in the instances with more robots.

8.3. Linear Arrays. We test on randomly generated linear arrays of length $n = k$ for $n = 1, \ldots, 1000$, and run ten tests for each length. The makespans produced by RIP are remarkably linear, and on average slightly less than the length of the array $n$. Curiously, RIP often produces makespans greater than $n$. For contrast, the sorting network implementation of parallel bubblesort given at [http://tinyurl.com/damu92](http://tinyurl.com/damu92) is designed to run in exactly $2n - 2$ timesteps. In fact, the algorithm can be terminated after only $n$ timesteps. To see why, recall that serial bubble sort works by launching bubbles from the first array entry so that they catch the largest entry and bring it to the end of the array, stopping before the numbers that have been brought to the end by earlier bubbles. Consider instead launching bubbles as soon as possible, rather than waiting for $n$ (or $n - i$) timesteps as in serial bubble sort. An indispensable graphic is given on slide 9 of [http://tinyurl.com/jegusez](http://tinyurl.com/jegusez). Assume the length of the array is even. We can imagine launching two new bubbles from either end of the array every two timesteps. For $i = 1, \ldots, n/2$, the bubbles launched on timestep $2i - 1$ eventually catch robots $i$ and $n + 1 - i$. After catching their robots, those bubbles do not release them until they make it to positions $n + 1 - i$ and $i$. Those robots have then made it to their goals, and will not be disturbed. The $i^{th}$ pair of bubbles finishes at timestep $(2i - 1) + (((n + 1 - i) - i) - 1) = n - 1$. Accounting for the possibility that $n$ may be odd gives the bound of $n$ timesteps as desired. Since we care about the robots getting all the way...
to their goals, rather than to the mid vertex, we think the bubbles give a nicer illustration than the case-based guarantees used in the proof of bubbletree.

It is also clear that at least $n$ steps are needed in the worst case. For example, if $n$ is odd and the two robots at the endpoints of the array both need to traverse the entire array to get to their goals.

![Figure 1](image)

Figure 1. Best, worst, and average makespans for the dense linear array tests.

| Robots | Density | RIP | Makespan | Runtime | Swaps | Optimal Flow | Makespan | Runtime | Swaps | Suboptimal Flow | Makespan | Runtime | Swaps |
|--------|---------|-----|----------|---------|-------|--------------|----------|---------|-------|---------------|----------|---------|-------|
| 10     | 0%      | 20.73 | 0.3s | 1.83 | 20.67 | 1.78s | 0.70 | 20.67 | 0.55s | 2.70 |
| 10     | 5%      | 21.00 | 0.03s | 1.97 | 20.77 | 1.61s | 1.07 | 20.77 | 0.53s | 2.37 |
| 10     | 10%     | 21.63 | 0s | 2.47 | 21.13 | 1.52s | 0.83 | 21.13 | 0.54s | 3.13 |
| 10     | 15%     | 22.60 | 0s | 2.27 | 22.37 | 1.65s | 1.57 | 22.37 | 0.64s | 3.30 |
| 10     | 20%     | 23.33 | 0.03s | 2.93 | 23.23 | 1.57s | 2.23 | 23.23 | 0.63s | 3.77 |
| 10     | 25%     | 24.80 | 0.03s | 3.63 | 24.70 | 1.52s | 3.50 | 24.70 | 0.70s | 4.97 |
| 10     | 30%     | 31.43 | 0.03s | 6.77 | 30.67 | 2.01s | 5.37 | 30.73 | 1.02s | 7.03 |
| 20     | 0%      | 24.00 | 0.5s | 8.17 | 23.67 | 5.20s | 6.83 | 23.67 | 2.05s | 9.87 |
| 20     | 5%      | 24.37 | 0.17s | 7.93 | 23.77 | 4.89s | 5.77 | 23.77 | 2.09s | 10.90 |
| 20     | 10%     | 24.93 | 0.07s | 9.30 | 24.07 | 4.61s | 6.50 | 24.07 | 2.19s | 11.67 |
| 20     | 15%     | 24.23 | 0.03s | 10.40 | 23.67 | 3.92s | 7.80 | 23.67 | 1.87s | 12.77 |
| 20     | 20%     | 26.23 | 0.07s | 13.27 | 25.20 | 3.88s | 9.37 | 25.20 | 2.00s | 15.20 |
| 20     | 25%     | 29.77 | 0.23s | 17.80 | 28.77 | 5.05s | 13.27 | 28.77 | 2.95s | 19.97 |
| 20     | 30%     | 34.50 | 0.1s | 24.13 | 33.03 | 7.20s | 20.80 | 33.03 | 3.97s | 26.30 |
| 30     | 0%      | 25.00 | 0.2s | 16.30 | 24.10 | 9.52s | 13.70 | 24.10 | 5.41s | 18.40 |
| 30     | 5%      | 26.33 | 0.07s | 18.50 | 25.63 | 10.81s | 16.80 | 25.63 | 6.48s | 22.10 |
| 30     | 10%     | 25.33 | 0.1s | 20.17 | 24.20 | 8.59s | 15.30 | 24.20 | 4.73s | 21.37 |
| 30     | 15%     | 26.23 | 0.13s | 23.97 | 24.97 | 8.07s | 19.00 | 24.97 | 4.71s | 24.47 |
| 30     | 20%     | 28.27 | 0.23s | 33.07 | 26.70 | 9.17s | 23.60 | 26.70 | 6.91s | 31.27 |
| 30     | 25%     | 33.23 | 0.07s | 43.60 | 30.27 | 22.16s | 30.37 | 30.27 | 22.90s | 38.73 |
| 30     | 30%     | 37.77 | 0.17s | 54.77 | 34.87 | 36.47s | 41.53 | 34.87 | 35.46s | 53.63 |
| 40     | 0%      | 26.43 | 0.13s | 30.20 | 25.13 | 23.78s | 23.83 | 25.13 | 13.31s | 31.93 |
| 40     | 5%      | 27.33 | 0.1s | 34.50 | 25.67 | 22.55s | 27.77 | 25.67 | 15.09s | 33.60 |
| 40     | 10%     | 27.50 | 0.07s | 39.00 | 25.40 | 20.76s | 31.67 | 25.43 | 14.25s | 38.53 |
| 40     | 15%     | 28.37 | 0.2s | 42.20 | 25.67 | 18.75s | 34.83 | 25.67 | 16.33s | 41.77 |
| 40     | 20%     | 29.73 | 0.3s | 55.03 | 25.33 | 22.27s | 39.53 | 25.33 | 19.67s | 44.70 |
| 40     | 25%     | 32.20 | 0.13s | 66.73 | 28.77 | 53.38s | 46.20 | 28.77 | 61.20s | 55.40 |
| 40     | 30%     | 39.73 | 0.2s | 92.10 | 33.20 | 130.33s | 68.63 | 33.20 | 92.78s | 79.53 |
| 50     | 0%      | 28.23 | 0.3s | 49.57 | 26.20 | 78.38s | 39.93 | 26.20 | 53.31s | 46.50 |
| 50     | 5%      | 29.20 | 0.3s | 53.03 | 26.07 | 61.78s | 40.93 | 26.07 | 49.57s | 51.73 |
| 50     | 10%     | 28.40 | 0.2s | 59.00 | 25.40 | 46.91s | 45.50 | 25.40 | 37.02s | 54.00 |
| 50     | 15%     | 30.47 | 0.23s | 71.60 | 26.73 | 68.54s | 55.43 | 26.73 | 56.52s | 62.20 |
| 50     | 20%     | 31.73 | 0.3s | 79.67 | 27.50 | 89.05s | 61.17 | 27.50 | 94.40s | 67.53 |
| 50     | 25%     | 37.70 | 0.4s | 109.50 | 30.33 | 259.08s | 75.50 | 30.37 | 268.41s | 86.83 |
| 50     | 30%     | 44.93 | 0.33s | 154.87 | 34.97 | 596.25s | 105.53 | 34.97 | 522.42s | 123.83 |
The runtime per timestep appears to be quadratic. In the worst case, we would expect a quadratic number of seconds per timestep – this could come from any of the inner while loop, resolving cycles, and swapping. Indeed

1. We could move \( \theta(k) \) robots, and have to loop through all \( k \) each time. This is improved to linear time in the optimizations section using inverse chains. (Of course, with \( k = n \), we only ever resolve cycles and perform swaps.)
2. We could build a length \( \theta(k) \) chain for each robot, only to have it terminate at a robot that has already moved or is at its goal. This is also improved to linear time using inverse chains.
3. It can take \( \theta(n) \) steps to check if two paths are subsets. For example, if both paths are length \( \theta(n) \) and only differ in one node. Presumably we could record swap risks and check when they might become swap needs, but we do not know a good way to do this.

![Figure 2. Square root runtime per timestep for the tests on dense linear arrays. Note that there are several small spikes present due to the fact that my computer was running other programs during the tests.](image)

8.4. **Square Arrays.** We test on randomly generated square arrays of length \( \sqrt{n} = \sqrt{k} \) for \( \sqrt{n} = 1, \ldots, 50 \), and run ten tests for each length. Superficially, the makespan graphs appear linear with the number of robots. However, RIP produces makespans that are consistently greater than the number of robots. In fact, inspecting the data suggests the makespans are superlinear. The makespan divided by the number of robots produces a nonconstant, and possibly logarithmic, curve. This is surprising for two reasons:

1. We could also form a hamiltonian path on the square grid. From the tests on linear arrays, we are convinced that this would produce a makespan slightly less than the number of robots.
2. Shearsort sorts square arrays in \( 2\lceil\lg n\rceil \sqrt{n} \) serial timesteps – much faster than for linear arrays.

More precisely, shearsort takes \( 2\lceil\lg n\rceil \) times the amount of time it takes to sort a single row, which is \( \sqrt{n} \) timesteps with parallel bubblesort. For the sake of simplicity, we ignore the ceiling in my graphs. See [http://tinyurl.com/zue7shp](http://tinyurl.com/zue7shp) for a brief description and proof of completeness for shearsort.
Figure 3. The first two graphs are a comparison between RIP and shearsort. The third graph shows that the makespan produced by RIP grows superlinearly.

The (square root) runtime per timestep is almost exactly the same as in the linear array tests. The square array tests are about 10% faster per timestep at the 1000-robot mark. This is directly offset by the fact that the square array makespans are about 10% larger at the 1000-robot mark.

Figure 4. Square root of the average runtime per timestep for the tests on the dense square arrays.

The number of swaps also appear super linear. However, it grows much more slowly than \((n^2)/2\) which is the expected number necessary swaps for dense linear arrays (every pair of robots has a 1/2 chance of being in the wrong order.)

Figure 5. The number of swaps appear to grow superlinearly but subquadratically.

It should also be noted that I had to manually remove fewer than ten outliers total for the average runtimes for the linear and square tests. I was using my computer while running
the tests, and there were several times when the test runtime exploded by several orders of magnitude while my computer processed a burdensome but unrelated task.

9. Optimizations

9.1. RIP Optimizations.

9.1.1. Inverse Chains. We can perform all the advances in the inner while loop of RIP in a single pass, and we can combine it with the cycle phase as well. The idea is to partition the robots into doubly linked maps from each robot to the robot occupying the node it wants to go to and vice versa. We can get the exact same behavior as in RIP as follows: Consider a single such map. Form a maximal list by starting at a random robot and going as far in the forward direction as possible. If going forward this way ends with a robot at its goal, then go on to the next inverse chain; that goalie robot will be swapped in the subset-swapping phase (or else that robot is the entire inverse chain). If going forward this way forms a cycle, then resolve the cycle. If going forward this way terminates at an empty node, then build a maximal chain by working backwards from the empty node, always selecting the robot whose index is the least greatest than the current robot’s index, mod $k$ (the empty node is treated as index 0.) Advancing that chain removes the need to loop. All told this only take $O(k)$ operations per timestep. For contrast, the original inner while loop in the basic implementation of RIP takes $\theta(k^2)$ runtime steps in the worst case, and checking for cycles can take $\theta(k^2)$ runtime steps as well (for example, if the inverse chain is linear).

We can also use inverse chains to get makespan optimizations. If there are no cycles in an inverse chain, then we potentially have many maximal chains to choose from. Intuitively, it makes sense to advance the longest chain. Another idea is to advance the chain with the robot that still has the farthest to travel. We can also build out a chain by starting at the empty node and, at each choice when working backwards, choosing the robot that still has the farthest to go. Choosing robots this way can have the advantage of eliminating swap risks. There are some maps in which it can cut the makespan in half.

For example, in the basic implementation of RIP, say all the robots are vying for the same node $v$, and the rest of their paths are linearly nested, with $P_{i,1}, \ldots, P_{i,\text{end}} \subset P_{i+1,1}, \ldots, P_{i+1,\text{end}}$. The proof of completeness did not depend on the order of the inner while loop, the subset-swaps, and resolving cycles. Imagine that we perform the subset-swaps before the inner while loop. How would this play out? First $r_1$ would enter, only to be subset-swapped by $r_2$ in timestep 2, which would be subset-swapped by $r_3$ etc. Finally $r_k$ would enter $v$ at time $k$, and then go off to $g_k$. Meanwhile, the entire process would have to start over again until $r_{k-1}$ could enter $v$. Overall, it would take $k(k - 1)/2$ timesteps just to resolve the massive bottleneck at $v$. Indeed, if the paths are made as short as possible, then this family of examples gives a $\theta(SIC + k^2)$ makespan with basic RIP. Of course, we could resolve it in $k$ steps by using any of the various inverse chain optimizations. If we perform the inner while loop before the subset-swaps, then small tests produce makespans between $2k$ and $3k$. So, compared to the baseline implementation of RIP, inverse chains do not give a dramatic improvement on small tests.

We have only implemented the basic version of RIP, however, and cannot compare what effect these optimizations would have on larger and more typical grid maps.

9.2. Bubbletree Optimizations.
Algorithm 3: RIP with Inverse Chains

1. for $i=1,\ldots,k$ do
   \[ P_{i,0} = \text{shortestPath}(s_i, g_i); \]

2. while some agent is not at its goal do
   3. partition robots into inverse chains;
   4. for each inverse chain do
      5. advance all the robots in one maximal chain;
   6. for $i = 1,\ldots,k$ do
      7. if $r_i$ is unmoved and there is an unmoved robot $r_j$ occupying $r_i$’s next node then
      8. if $P_{j,t} \subseteq P_{i,t}$ then
         9. swap($r_i, r_j$);

9.2.1. Forming the tree. There are various characteristics that we might want the bubbletree to have:

   1. Distance “preserving”, especially between pairs of starts and goals
   2. Low diameter
   3. Low number of nodes

   Unfortunately, it is impossible to restrict to a tree while preserving distances to within a constant factor, as is demonstrated by polygon graphs.

   (Hassin and Tamir 1995) presents a polynomial method to form a spanning tree of (nearly) minimum diameter. Simply form a breadth-first-search tree from the “one-center”, the node whose max distance $d$ to all other nodes is minimized. It is clear that the minimum diameter must be at least $d$. And the breadth-first-search tree has diameter at most $2d$.

   For minimizing the number of nodes, it is useful to use Steiner trees. For a weighted-graph $G$ with a subset $S \subset V$ of the vertices, the generalized Steiner tree problem is to find a subtree $T$ containing all the vertices in $S$, such that the sum of the edge weights is minimized. It is possible to approximate the minimum to within a factor of $\ln(4)$ (Byrka et al. 2010), although approximating to within a factor of $96/95$ is NP-hard (Chlebík et al. 2008).

9.2.2. Bubble Forest. We can get an even bigger improvement, especially on maps with relatively few robots, by forming a bubble forest. (Lopez 2006) presents a Steiner forest, which can be found in polynomial time by solving a dual linear program and is able to

   • Ensure each $\{s_i, g_i\}$ pair is in the same tree
   • minimize the total number of edges to within a factor of 2 (in fact, the algorithm works more generally for edges with positive-rational length.)

   Of course, minimizing the number of edges also does a lot to minimize the number of nodes, as the typical PERR formulation is on a graph with unit-cost edges.

9.2.3. Goal-balancing $v_m$. To get a linear makespan, we need to show that the problem shrinks in size by a constant factor (half) with each recursive call. In the basic bubbletree algorithm, the main measure of the problem size is the number of nodes. To prove the $O(k+l)$ result, however, we need to express the problem size in terms of $k$. Thus we choose
the mid node \( v_m \) such that each of the subtrees defined by its children has at most \( k/2 \) goal vertices. The procedure is very similar to that in basic bubbletree: perform a breadth-first search from an arbitrary vertex \( v \), and mark every vertex \( v' \) with the number of goal vertices in the subtree it defines, \( g(v') \). If \( g(c) \leq k/2 \) for every child \( c \) of \( v \), then we are done. Otherwise let \( d \) be the unique goal node for which \( g(d) > k/2 \). Then \( d \) is a satisfactory mid node.

9.2.4. Moving all Migrant Sets. In the base algorithm, we perform two main (non-recursive) operations: advancing the focal \( M_c \) in priority order, and advancing a chain in \( G^T_c \) with root \( v_m \). We can simultaneously move other migrants, subject to ceding priority to the focal \( M_c \) and then chain, and maintaining the priority orderings within each of the migrant sets.

Refer to bubbletree with the joint optimizations of goal-balancing together with moving all migrant sets as bubbletree2.

**Proposition 9.1.** Bubbletree2 produces a makespan of \( O(\text{diam} \ast \log k + k) \).

**Proof.** The proof uses the following lemma.

**Lemma 9.2.** After \( t \) timesteps, the \( i^{th} \) highest priority robot in any \( G^t_c \) (the highest priority is \( i = 0 \)) is already an infiltrator or else is at most \( \max\{\text{diam} + 2i - t + 2k, i+1\} \) steps away from \( v_m \).

**Proof.** \( 2i - t \) is as in the proof of the guarantee for the basic bubbletree algorithm, \( 2k \) comes from the fact that we spend at most \( 2k \) timesteps moving \( r_m \), and \( i + 1 \) comes from the new possibility that a line of migrants is waiting for \( v_m \) to empty. We can thus ignore the \( 2k \) term. The induction proceeds with the exact same cases as in the proof of the guarantee for basic bubbletree. Clearly the base case holds. We will reproduce the rest of the argument here.

Assume we have the stated invariant at timestep \( t \) (and ignore the \( 2k \) term). It is still the case that infiltrators are never pulled back across \( v_m \). Consider what it means if some migrant \( r_i \in M_c \) that has not yet gotten to \( v_m \) fails to advance. There are the same two old possibilities, and also a new possibility. The first is if a higher priority robot \( r_j \) swaps another robot to the node that \( r_i \) was planning on advancing to. Then \( r_i \) is only one step worse off than \( r_j \) was the turn before at \( t-1 \). The guaranteed proximity to \( v_m \) for \( r_j \) was \( \max\{\text{depth}(T_c) + 2(i - 1) - (t - 1), (i - 1) + 1\} = \max\{\text{depth}(T_c) + 2i - t - 1, i\} \). So the guarantee for \( r_i \) is \( \max\{\text{depth}(T_c) + 2i - t, i + 1\} \) as desired. The second is that a higher priority robot \( r_j \) bully-swapped it. This bodes even better for \( r_i \) than if it is smushed, and the guaranteed closeness to \( v_m \) is again maintained. The new possibility is if \( r_i \) is stymied by the robot occupying \( v_m \). Then \( r_i \) is at most one step from \( v_m \), and we are good. \( \square \)

By the lemma, every migrant is within \( k \) of \( v_m \) after \( \text{diam} + 4k \) timesteps. We can then invoke the basic bubbletree guarantee, since the optimization honors the priorities from basic bubbletree. The effective depth of each subtree \( T_c \) is at most \( |M_c| \), so we can finish off the pre-recursive phase in at most \( \sum_c |M_c| + \sum_c 2|M_c| + 2k \) more steps (recall the guarantee from the basic analysis is \( \text{depth}(T_c) + 2|M_c| \) per tree \( T_c \), plus \( 2k \) steps for \( r_m \)). So the pre-recursive phase takes at most \( \text{diam} + 10k \) timesteps.

When we recurse, the number of robots decreases by at least half. Unfortunately, we have no such guarantee on the diameter (or the longest path length). Thus we solve the recursion \( f(\text{diam}, k) \leq O(k + \text{diam}) + f(\text{diam}, k/2) \) and get \( f(\text{diam}, k) = O(\text{diam} \ast \log k + k) \) as claimed.
9.2.5. Advancing Infiltrators. We can also advance robots that are already in their target trees. We refer to these non-migrants as infiltrators (they include all non-migrants other than $r_m$.) Since the infiltrators have the lowest priority, it is not very important how to move them. A simple solution is to initialize an ordering on each $G^T_c$ and move the infiltrators in priority order. A more complicated approach is to try move the infiltrators in $T_c$ out of the way of the migrants in $T_c$.

What we see as the most natural approach is to preemptively do the actions that would be taken after the pre-recursive phase. Let $T^i_c$ denotes a subtree in an $i^{th}$ recursive call of bubbletree. Thus there is one $T^0_c$, and it is the entire bubble tree. The $T^1_c$‘s are the familiar subtrees defined by the children of $v_m$. We say a $T^i_c$ is order $i$. We can similarly define the higher order migrant set $M^i_c$ as the robots in $T^i_c$ whose goals are not in $T^i_c$. This way we have $M^i_c \subset M^{i+1}_c$ when $c'$ is a descendant of $c$. In each timestep, loop through the $T^i_c$’s from lowest to highest order, and move the robots in $T^i_c$ as in bubbletree2, treating robots in strictly lower-order migrant sets as immovable obstacles. There are only $O(k)$ $T^i_c$’s, so this optimization is not excessively time-consuming.

Using the additional optimization of advancing infiltrators to bubbletree2 gives an $O(k+l)$ solution to the counterexample that proved the necessity of the log$k$ factor in bubbletree2. However, we do not know if it gives an $O(k+l)$ solution in all cases. In particular, the longest path to goal (or the diameter) need not decrease by any constant factor in the first recursive call. For example, they would not decrease if there was an order 1 robot $r_1$ whose path to goal was $l$ (with $l$ nearly the size of the diameter): we could start with $r_1$ at a grandchild of $v_m$, and engineer it so that there is always an order 0 robot occupying that grandchild’s parent, or else vying for it.

9.2.6. Recomputing Bubble Trees. When we recursively call bubble tree on a subtree, we can instead pass the nodes of the tree and compute a new bubble tree (or forest) on those nodes.

9.2.7. Earlier Recursive Calls. We can also call bubbletree on each $T_c$ as soon as the robots occupying $T_c$ are precisely the robots in $G^T_c$.

9.2.8. Leafhunter Hierarchy. We can be intelligent about how we assign priority within each $G^i_c$ (and each $G^T_c$), rather than doing it randomly. We can require higher priority for robots that will have farther to go after becoming infiltrators. More minimally, we can insist on higher priority for $r_i$ than $r_j$ if $g_i$ is a descendant of $g_j$.

Using the leaf-hunter hierarchy also gives a means for smarter chain resolution. In the base algorithm, we choose the chain to advance randomly. If we prioritize leaf hunters, then at least the structure of these chains will not be completely random. Rather, the deeper members of these chains are more likely to be leaf-hunters, and therefore be relatively better off heading the push into the depths of the subtree.

9.2.9. Postprocessing. With some of these more complicated optimizations (in particular, advancing infiltrators), there is a risk that we introduce redundant movement. For example, we may advance an infiltrator up to $v_m$, only to have it swapped back the next turn by a higher-priority migrant. There is a redundancy precisely when $r_i$ occupies $v$ at times $t$, vacates $v$ at time $t + 1$, and returns at time $t + c$ without any other robot occupying $v$ in the meantime. A general path-planning optimization is to go through in postprocessing
and replace this redundant traveling with waits (cite the push and swap paper.) It may be necessary to loop through the solution multiple times, as new moves may become redundant after deleting old redundant moves.

10. Comparison Between RIP and BubbleTree

RIP is easier to implement. RIP can also outperform bubbletree on cyclic graphs e.g. on the cycle graph at the end of the basic bubbletree makespan analysis, a graph on which RIP performs optimally. RIP also performed near-optimally in the DAO map, the 20x15 grid maps, and the dense linear arrays.

Extrapolating RIP’s performance on the dense square grids suggests that bubbletree would outperform RIP on robot-dense two-dimensional arrays. Bubbletree also has much stronger theoretical guarantees.

11. Discussion

We went through a lot of work to show that RIP produces a makespan that is $O(SIC+k^2)$. Empirically, RIP performs much, much better. And we have not found any examples that are $O(SIC+k^2)$ but not $O(k^2)$. In fact, it could still be that optimizations of RIP obtain an $O(n)$ or even an $O(k+l)$ makespan! (Where $l$ is the maximum distance from start to goal.)

Why is it so difficult to prove a stronger result? It is very natural to modify the potential function. For example, we could include terms for the number of times pairs of paths intersect, or we could somehow amortize how often a robot is stymied by another robot, including stymied by a robot at the end of its chain. A natural conjecture is that $r_i$ only swaps $r_j$ once. But we can in fact engineer scenarios in which $r_i$ swaps $r_j$ arbitrarily many times. It is sufficient to show that we can pull $r_i$ and $r_j$ off of their goals and force them to subset-swap. To manage this, start with $r_i$ at (0,0) and $r_j$ at (1,0) and with two robots above $r_j$. Have those two robots subset-swap $r_j$ up to (1,2), and then have a stream of robots file through (1,1), preventing $r_j$ from moving. We can then have three robots pull $r_i$ over to (3,0). We can then free $r_j$ to its goal. But now there is a swap need ($i,j$). How to amortize these machinations is beyond me.

Maybe $r_i$ can keep a labeled tally of the times it has been stymied, and share that tally with any robot it subset-swaps. Intuitively it makes sense that, if $r_i$ swaps $r_j$ at time $t$, then $r_j$ is at least as well off at time $t+1$ as $r_i$ was guaranteed to be at time $t$. But what exactly to tally, and how to tally it, is unclear.

Another approach is to use inverse chains. We can modify RIP so that the largest subchain of an inverse chain is guaranteed to advance every other timestep. But what if the trees are shallow and branch rapidly? Another approach is to select the subchain containing the robot that still has the farthest to go – this way the maximum distance to goal can be tracked and shown to consistently decrease. But, there can be many robots that are the same distance to goal, and we can only be guaranteed to advance one of them – and only every other turn, at that.

Really what we need is a linear measure of global progress. Of course, this is the motivation for trying a different algorithm, like bubbletree.
12. Acknowledgments

Thank you to USC’s SURE program for funding this research. Thank you to Guni Sharon for making the code of his CBS solver available to us. Thank you to Sven Koenig and Hang Ma for working with me in their lab, including: presenting me with PERR and the pseudocode for RIP, as well as the idea of using a monovariant; giving me an overview of their prior research; helping with the code and Excel; talking through my proofs and ideas with me; and securing funding through SURE.

References

[1] Ajtai, M., Komlos, J., Szemerdi, E., An O(nlogn) Sorting Network. Published in: STOC ’83 Proceedings of the fifteenth annual ACM symposium on Theory of computing. Pages 1-9.
[2] Byrka, J.; Grandoni, F.; Rothvo, T.; Sanita, L. (2010). An improved LP-based approximation for Steiner tree. Proceedings of the 42nd ACM Symposium on Theory of Computing. pp. 583-592.
[3] Chlebík, Miroslav, and Janka Chlebkov. The Steiner tree problem on graphs: Inapproximability results. Theoretical Computer Science 406.3 (2008): 207-214.
[4] Coltin, B., and Veloso, M. 2014. Scheduling for transfers in pickup and delivery problems with very large neighborhood search. In AAAI Conference on Artificial Intelligence, 2250-2256.
[5] Haberman, N. (1972) Parallel Neighbor Sort (or the Glory of the Induction Principle), CMU Computer Science Report (available as Technical report AD-759 248, National Technical Information Service, US Department of Commerce, 5285 Port Royal Rd Sprigfield VA 22151).
[6] Hassin, R., and Tamir A. 1995. On the minimum diameter spanning tree problem. Inform. Process. Lett., 53 (2), pp. 109-111 Kornhauser Kornhauser, D.; Miller, G.; and Spirakis, P. 1984. Coordinating pebble motion on graphs, the diameter of permutation groups, and applications. In Annual Symposium on Foundations of Computer Science, 241-250.
[7] Lang, Sorting on two-dimensional processor arrays H.W. tinyurl.com/zue7shp. Created: 29.11.2004. Updated: 26.01.2008. Date of Access August 20, 2016.
[8] Lopez, Adriana. Steiner Trees and Forests. Massachusetts Institute of Technology 18.434: Seminar in Theoretical Computer Science. tinyurl.com/jonczgr. Published March 7, 2006. Date of August 20, 2016.
[9] Luna, R., and Bekris, K. 2011. Push and swap: Fast cooperative path-finding with completeness guarantees. In International Joint Conference on Artificial Intelligence, 294-300.
[10] Ma, H.; Tovey, C.; Sharon, G.; Kumar, T. K. S.; and Koenig, S. 2016. Multi-Agent Path Finding with Payload Transfers and the Package-Exchange Robot-Routing Problem. In AAAI Conference on Artificial Intelligence.
[11] Muhammad, Rashid. Bubble Sort tinyurl.com/damm92. Date of Access August 20, 2016.
[12] Sharon, G.; Stern, R.; Felner, A.; and Sturtevant, N. 2015. Conflict-Based Search for Optimal Multi-Agent Pathfinding. Artificial Intelligence 219:40-66.
[13] Silva, Fernando. Parallel Algorithms – Sorting. tinyurl.com/jeguez. Date of Access August 20, 2016.
[14] Sturtevant, N. 2012. Benchmarks for Grid-Based Pathfinding. Transactions on Computational Intelligence and AI in Games 4(2):144 - 148.

Department of Mathematics, Dartmouth College
E-mail address: james.r.drain.17@dartmouth.edu