Bounds on multi-agent map exploration (and a game called exapunks)

Josh Vander Hook
Multi-Agent Reading Group, 2020-04-23
This talk

Mainly presenting:

*Tight bounds for undirected graph exploration with pebbles and multiple agents*

Yann Disser, Jan Hackfeld, and Max Klimm
disser@mathematik.tu-darmstadt.de
{jan.hackfeld|max.klimm}@hu-berlin.de
Nov 2018 (arxiv); Jan 2020 (Journal of ACM)

• How big a program does it take to explore a given maze?
• How much do you have to remember as you explore?
• How much do additional agents help?
• *Why is it so expensive to solve POMDPs?*
Preliminaries

- Undirected graphs (at first, with same degree on each node)
- Though, first designed for mazes
- No id on nodes
- Local ports are numbered (not edges)
- Agent: State transition function
  \[(p_g, p_r, \delta, s)^+ = f(p_g, p_r, \delta, s)\]
- Graph exploration
  - Verification of programs
  - Accepting a grammar or string or language
  - Decidability of certain optimization problems
- S-T connectivity:
  - Web search and indexing

Q: How much information about the maze is required to explore it?
How to judge an agent

- Memory usage (in bits)
- Steps to explore a graph
- Generality (which graphs are ok?)
- Algorithmic states required
- Pebble count
Exapunks is essentially “right”

- Port Numbers
- Agents
- Files ~= pebbles
- Memory
<table>
<thead>
<tr>
<th>M=</th>
<th>State</th>
<th></th>
<th># Pebbles</th>
<th># Agents</th>
<th>Can / Cant</th>
<th>Graph</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>K</td>
<td></td>
<td>1</td>
<td>Cannot</td>
<td>All Mazes</td>
<td>Budach</td>
<td></td>
<td></td>
</tr>
<tr>
<td>K</td>
<td>2</td>
<td>1</td>
<td>Can</td>
<td>All Mazes</td>
<td>Blum</td>
<td></td>
<td></td>
</tr>
<tr>
<td>K</td>
<td>K</td>
<td>Cannot</td>
<td>Planar</td>
<td>Blum</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>\log n</td>
<td></td>
<td>1</td>
<td>$n^3 \log n$ (random)</td>
<td>Regular</td>
<td>Aleliunas</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>$Dn^2$</td>
<td>Undirected</td>
<td>Dudek</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>$c^n$ (random)</td>
<td>Directed</td>
<td>Aleliunas</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Unlimited</td>
<td>1</td>
<td>1</td>
<td>$O(n^8)$</td>
<td>Directed</td>
<td>Bender</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$\Omega(D\log n)$</td>
<td>$O(n)$</td>
<td>1</td>
<td>$c^n$</td>
<td>Any</td>
<td>Fraigniaud</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$O(D\log d)$</td>
<td></td>
<td>1</td>
<td>Any</td>
<td>Fraigniaud</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$O(n^2 d_m \log n)$</td>
<td>$O(\log \log n)$</td>
<td>1</td>
<td>$n^c$</td>
<td>Directed</td>
<td>Fraigniaud</td>
<td></td>
<td></td>
</tr>
<tr>
<td>*</td>
<td>$\Omega(\log \log n)$</td>
<td>1</td>
<td>$n^c$</td>
<td>Directed</td>
<td>Bender</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$k$</td>
<td>$\Theta(\log \log n)$</td>
<td></td>
<td>$n^c$</td>
<td>Undirected</td>
<td>This</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$\Theta(\log \log n)$</td>
<td>$\Theta(\log \log n)$</td>
<td>1</td>
<td>$n^c$</td>
<td>Undirected</td>
<td>This</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Single agent upper bound (trees vs graphs)

- Pick an edge.
- Next time, other edge, so that’s 1 bit ($\log d$) to store per node
- Traverse down until no edges
- Go back up (need to store “back” edge)

In order walk: S=0,0,2,1,2,2,1,0,2,1,2,2
S can be generated for any full d-ary tree

Hold this thought: Could we generate $S$ without actually knowing the graph?
Single agent upper bound (trees vs graphs)

- Pick an edge.
- Next time, other edge, so that’s 1 bit ($\log d$) to store per node
- Traverse down until no edges
- Go back up (need to store “back” edge)

Nice tree:

1. A counter for depth
   - worst case $D = \log n$
2. A set of “next edge” counters $D \times \log(d)$
Single agent upper bound (trees vs graphs)

- Pick an edge.
- Next time, other edge, so that’s 1 bit \((\log d)\) to store per node
- Traverse down until no edges
- Go back up (need to store “back” edge)

Nice tree:
1. A counter for depth
   - worst case \(D = \log n\)
2. A set of “next edge” counters \(D \times \log(d)\)

Bad Tree (Trap)
1. Drop a pebble in explored nodes
Single agent upper bound (trees vs graphs)

**Nice tree:**

1. A counter for depth
   - worst case $D = \log n$
2. A set of “next edge” counters $D \times \log(d)$

**Bad Tree (Trap)**

1. iteratively assume $D$ to not get stuck

---

- Pick an edge.
- Next time, other edge, so that’s 1 bit ($\log d$) to store per node
- Traverse down until no edges
- Go back up (need to store “back” edge)

[as noted by Fraigniaud 05]
Traps

- Traps are a common technique to prove lower bounds on required agents, memory, and/or state size against any exploration algorithm.
- More trap examples are easily found in references, but they are too long to present here.
- In fact, there exists a trap for any finite set of agents with finite memory. [Fraigniaud]
  - It’s big. $O(s \uparrow \uparrow (k + 1))$ for $k$ agents with state size $s$.
- ACM paper drastically reduces that upper bound!
What about randomization? [Aleliunas 79]

- Let’s prove the number of steps to explore in expectation
- Defines a Markov process $p_2 = Mp_1$ with matrix $M_{i,j} = 1/d_j$ if $e_{i,j} \in G$
  - (just adjacency matrix / degree)
- so we can solve for the steady-state ... $p_i = d_i/2e$, which implies in a random sequence, the mean time between visits to a node is $2e/d_i$
- ... so probability of edge $e_{i,j}$ being taken in any traversal is $\frac{d_i}{2e} \times \frac{1}{d_i} = \frac{1}{2e}$,
- ... An edge from an adjacent node back to start is expected to be covered after about $2e$ transitions
- If we fix a tour (say, a minimum spanning tree), and add up the total number of adjacent transitions back, we have $\leq 2e(n-1)$ expected transitions to follow that tour (includes many “wasted” transitions).
- For $d$-regular graphs, it’s half that $e = \frac{1}{2}(d \times n)$
What about randomization? [Aleliunas 79]

- Unfortunately, a “combination lock” graph requires exponential time to explore with randomized algorithms
  - (Combination refers to the sequence of 0/1 labels to move “right” above.)

Cite: Blum
First remarkable result [Aleliunas 79]

There exists a sequence of edge transitions, calculated in advance, such that any \( d \)-regular undirected graph of size \( n \) will be explored.

1. Generate a random transition sequence \( S \) of length \( O(n^3 \log n) \)
   1. They consider this \((dn+1)\log n\) trials exploring the graph with sequences of length \( s = 2dn(n - 1) \)

2. Recall that \( P(t > a) \leq \frac{E[t]}{a} \frac{dn(n - 1)}{2dn(n - 1)} \)

3. So, \( p \) failure on any particular graph is \( \leq 2^{-(dn+1)\log_2 n} \leq n^{-(dn+2)} \)

4. Meaning \( n^{(dn+2)} \) graphs are required before we expect a failure

5. And there are “only” \( n^{(dn+1)} \) regular graphs (or, expected number of failures is \( n^{-(dn+2)} n^{(dn+1)} = \frac{1}{n} \))

This proof technique using indicator variables for failures is by Erdos.
Universal Traversal Sequence

- Works on any d-regular graph (but requires $\log(n)$ memory to compute and $\log(n)$ memory to keep track of # of steps)
- Also provides bounds on pursuit-evasion strategy (see same paper)
- Kicked off 40 years of research on derandomized universal sequences
- Very limited in terms of graphs they apply to, and amount of memory required to compute them
- Extremely limited in terms of actual sequences
- Extremely powerful for proving performance of algorithms or tackling decision problems
- Current state of the art: 3-regular universal exploration sequence by [Reingold 2008]

Example: In order walk $S=0,0,2,1,2,2,1,0,2,1,2,2$

S can be generated for any full d-ary tree
Adding in pebbles

An agent with one pebble and finite memory cannot explore all graphs [Hoffman 81]

Finite mazes are explorable with 2 pebbles [Blum 78]

Several combinations of agent models and graph types exist … See paper for more comprehensive review

Pebbles are usually dropped to mark nodes so they are distinguishable, esp in cases where $n$ is not known
On to the paper

- $\Theta(\log \log n)$ pebbles are necessary and sufficient to explore all undirected anonymous graphs with at most $n$ vertices for a single agent
  - Four ground breaking results: Necessary (lower bound), sufficient (upper bound), and extensions to all graphs with unknown $n$
  - Polynomial number of steps (as good as the best known)

- $\Theta(\log \log n)$ agents are necessary and sufficient to explore all undirected anonymous graphs with at most $n$ vertices

- For both lower bounds, they provide a “trap” graph with remarkably lower size graphs ($s^{2^{5k}}$)

(Note, I will not make use of their notation)
Lemma 1. Let $A$ be an agent with $s$ states and $p$ pebbles exploring a set of graphs $G$. Then there is an agent $A'$ with six states and $p + \lceil \log s \rceil$ pebbles that reproduces the walk of $A$ on every $G \in G$ and performs at most three edge traversals for every edge traversal of $A$.

Use $\log s$ pebbles (green) to encode agent $A$’s state.

Always keep green pebbles separate

$$(p_g, p_r, \delta, s)^+ = f(p_g, p_r, \delta, s)$$
Agents are as powerful as pebbles

Lemma 2. Let $A$ be an agent with $s$ states and $p$ pebbles exploring a set $G$ of graphs. Then, there is a set $A = (A_0, \ldots, A_p)$ of $p+1$ agents, where $A_0$ has $s$ states and all other agents have two states, that reproduce the walk of $A$ in every graph $G \in G$. Moreover, for every edge traversal of $A$ each agent in $A$ performs at most one edge traversal.

- Have agents *follow along*, pretending to be pebbles, and sit tight until someone comes to pick them up.
- They change their state by reading the state of the agents in the local node only (zero coms)
Exploring Graphs

- Unfortunately, the days of clever exploration algorithms seem to be behind us
- The proofs ahead all make use of the UXS to generate an exploration, and work to make that UXS applicable to more and more graph types
Exploring graphs

• Lemma 1: You can perform a closed walk over a subset of the graph by applying a Turing machine that modifies a universal exploration sequence for size $z \leq n$
  – Because they use UXs, they are limited to 3-graphs
  – They modify the UXS to “mirror it” $S + S'$ to get a closed walk
  – They prove this does not require more memory than a normal UXS (limited by counter to follow and log $z$ to generate)
  – They prove this does not require more than $z^c$ steps for small constant

Helpful to think of this Turing machine as a log(n) memory program that uses no pebbles and will walk over a fixed-size graph as long as the degree at every vertex is 3
Lemma 2: How to construct a 3-graph from any graph

- Use memory to simulate the previous exploration algorithm on this “new” graph. The output node is mapped back to the actual graph.
- … as you enter a node, observe out-degree, you can construct the above graph to get the right transition from the UXS.
- Prove a constant number of nodes are added at each node ($3 \times d_i$).
- Prove that a constant amount of memory is required to track the new subgraph.

Figure 1: Example for the transformation of a graph $G$ to a 3-regular graph $G_{reg}$. A vertex $v$ of degree 2 is transformed to a cycle containing 6 vertices and for the edge $\{v, w\}$, three edges are added to the graph.
Exploration Algorithm (with pebbles)

- \( f(p_g, p_r, \delta, s) \) is the new state transition
- \( s = g(h, m, Q) \) is the state of the internal Turing machine
- Call it an \((s, p, m)\)-machine for states, pebbles, and tape-length

- Yes, they will use the Turing tape.
Turing Machine Reductions

An \((s, p, m)\)-machine can be reproduced exactly by an \((cs, p + c, \frac{m}{2})\)-machine

Things go **off the rails** here …

- Select a walk through \(z\) of the nodes (Lemma 1, this is doable with \(\log z\) space)
- Consider *those nodes* an additional tape for a *meta-Turing machine, of which the agent is the head*. Pebbles are the marks.
Intermission: XKCD 505

Exploration Algorithm (with pebbles)

Magic reduction machine algorithm
1. Generate and follow a walk of size $2^m$ [Lemma 1]
2. If we run out of nodes: Stop, we have succeeded in exploring and cannot find the walk anyway. Return to origin with all pebbles
3. If graph is larger than walk, proceed by using some of the pebbles to encode the state of the “better” machine directly on the environment. Each state transition requires at most $2^{O(m)}$ transitions.
Environmental Computation

<table>
<thead>
<tr>
<th>$p_0$</th>
<th>$p_1$</th>
<th>$p_2$</th>
<th>$p_3$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

(a) Tape memory

Figure 2: Memory encoding by pebbles on a closed walk. The state of the tape of length $2m = 12$ in (a) is encoded by the position of the $c - 3 = 4$ pebbles in (b). The number of the vertices corresponds to the order of first traversal by the closed walk $\omega$ starting in 0. The position of each pebble encodes $m_1 = 3$ bits.

The “bootstrap” procedure of how to count unique nodes in $W$ (sf b), manage the memory on the tape (sf a), etc is spelled out in the paper as actual algorithms. **The effectively define malloc for pebbles + graphs**
• Counting unique nodes: Push a pebble forward, then backtrack to see if you see it again on the way back.
• If you get back to memory location 0, find your counter and increment it
• Follow the walk again until you find your pebble

![Algorithm 4](image-url)
Example: Reading and writing a bit

\textbf{Algorithms 6} Reading and writing one bit for the simulated memory.

\begin{algorithm}
\begin{algorithmic}
\Function{READBIT}{ }
\State $i \leftarrow \lfloor \frac{R_{\text{head}}}{m_1} \rfloor$
\State $j \leftarrow R_{\text{head}} - m_1 \cdot i$
\State id $\leftarrow \text{GETPEBBLEID}(p_i)$
\State \Return the $j$-th bit of id (in binary)
\EndFunction

\Function{WRITEBIT}{b}
\State $i \leftarrow \lfloor \frac{R_{\text{head}}}{m_1} \rfloor$
\State $j \leftarrow R_{\text{head}} - m_1 \cdot i$
\State id $\leftarrow \text{GETPEBBLEID}(p_i)$
\If{$b = 1$ and \text{READBIT}() = 0}
\State id $\leftarrow id + 2^j$
\ElsIf{$b = 0$ and \text{READBIT}() = 1}
\State id $\leftarrow id - 2^j$
\EndIf
\State \text{PUTPEBBLEATID}(p_i, id)$\end{algorithmic}$
\end{algorithm}
Mad Pebble Machine

- We can recursively simulate larger sequences [using the memory we have written into the graph]
- Call this an expansion
- Recall: An \((s, p, 2m)\)-machine can be reduced to an \((cs, p + c, m)\)-machine and walks out \(2^m\) nodes on initialization

- If \((sc^i, p + ic, m2^{-i})\) walks \(2^{m/2^i}\) nodes on initialization, for the \(i^{th}\) reduction
- Then \((s_0c^{-i}, p_0 - ic, m_02^i)\) walks \(O(2^{2^i})\) nodes on initialization, for the \(i^{th}\) expansion
- Therefore, first \(\text{sigmem}\) is when \(i = \log \log n\), because then it tries to walk too many nodes

Therefore, \(O(\log \log n)\) pebbles are used, as well as \(c^i = O(\log \log n)\) memory (subtlety here, that large #s of pebbles are not required if the \text{malloc} fails)
Not Covered

• Since agents are as powerful as pebbles, at most \(\log \log n\) agents are required to explore any graph

• A recursive graph construction algorithm is shown that can confound any set of agents with insufficient memory or pebbles (over 10 pages or so)
Recap

• In the 70s, we learned about universal traversal sequences
• In the 90s-2000s, that became universal exploration sequences
• Lately, those have been applied to ensure an exploration strategy exists for a few classes of graphs
• This year, it was extended to all graphs by this paper and drastically lowered the on-agent memory or # of agents required.
• The main insight was to leverage the graph itself as extra memory!
  • An agent with \( x \) “extra” pebbles can ”act like” an agent with \( 2^x \) memory and explore a graph with \( 2^{2^x} \) nodes.
  • Eight “compute” pebbles = explore the known universe’s atoms
• BUT it does require the ability to bootstrap and build (or call in) additional resources as it iterates and “guesses” larger graphs
Exapunks