A Polynomial Complexity "Snakes and Ladders Heuristic" for the Hamiltonian Cycle Problem

V. Ejov
and
S. Rossomakhine
Flinders Mathematical Sciences Laboratory
School of Computer Science, Engineering and Mathematics
joint work with
P. Baniasadi, J. A. Filar and M. Haythorpe,
all Flinders University
http://fhcp.edu.au

Hamiltonian Cycle, Traveling, Salesman and Related Optimization Problems Workshop, Flinders University, December 14-15, 2012

We also acknowledge private communications with:

B. McKay from ANU, G. Royle from UoWA, K. Helsgaun from Roskilde University, Denmark.

Because **SLH** classifies edges into two *dynamic* types we were reminded of the ancient “snakes and ladders” board game; hence the name.
"Snakes & Ladders" board game
Let $G$ be an undirected graph with $n$ vertices (and no self-loops) and $C$ be a given circle.

A “Slither” is the equivalence class of arrangements of vertices on the circle.

The edges of the graph are represented as either arcs, for edges between vertices that are next to one another, or chords, for all other edges.
Slithers on a graph

- An edge that is an arc on $C$ will be called a “snake”.
- An edge that is a chord on $C$ will be called a “ladder”.
- Two adjacent vertices on $C$ that are not joined by a snake form a “gap”.
- A Hamiltonian cycle is simply a slither of $n$ snakes with 0 gaps.
- Solving HCP by SLH: close all gaps on $S_G$ by means of designed transformations.
Example of a Slither

Flinders Hamiltonian Cycle Project - http://fhcp.edu.au
Snakes and Ladders Heuristic
There are two generic isomorphisms from which all other transformations used by SLH may be constructed.
Transformation to HC

• For any initial slither $S_G$ and any Hamiltonian slither $H_G$ there exists a transformation $T : S_G \leftrightarrow H_G$ of the form

$$T = \prod_{i=1}^{\ell} \gamma^{\varepsilon_i} \zeta^{\delta_i},$$

where $\varepsilon_i$ and $\delta_i$ take values 0 or 1 and $\ell$ bounds the number of transformations.

• SLH attempts to iteratively build such a transformation $T$ to some unknown Hamiltonian slither as a combination of $\gamma(.)$ and $\zeta(.)$ transformations.

• To ensure SLH is polynomially bounded and efficient, we choose $\ell \leq n^4$. 
SLH closing 2–opt type 1 transformation, $\gamma(y, x, a)$:
SLH closing 3-opt transformation, $\gamma(c, y, b) \circ \gamma(y, x, a)$:
Operations. SLH-floating operations

SLH floating 2–flo transformation, $\gamma(y, x, a)$:
SLH floating 4–flo type 1 transformation,
\( \gamma(d, b, y) \circ \gamma(b, e, f) \circ \kappa(x, a, c, d) \):
SLH floating 5-flo transformation, $\phi(j, d, g, h) \circ \phi(x, a, c, d)$:

![Diagram showing the transformation process](image-url)
SLH opening 4-flo transformation $\kappa(x, a, b, c)$:

Note: all transformations are designed to be performed for a slither containing a gap $(x, y)$, or, as we say, around gap $(x, y)$. 
Stage 0. Only SLH- closing operations are performed until no more closing operation is possible.

Stage 1. SLH-floating operations are performed. Emerging slithers, identified by an emerging gap, are stored in the slither list. If a closing operation becomes possible, it is performed, the slither list is deleted and stage 1 is repeated from scratch.

Stage 2. An opening operation is performed followed by floating operations. If \( n \) slithers are considered, then stage 2 is repeated by performing a different opening operation until all possible opening operations are considered.

Stage 3. After one opening operation is performed, we add the obtained slither to the slither list and then attempt floating transformations that close a gap without retreating to already listed slithers. If no floating transformation that closes a gap is available, then repeat the above by performing another opening operation and then floating operation that closes a gap.
Algorithm

- **Stage 0.** Only SLH-closing operations are performed until no more closing operation is possible.
- **Stage 1.** SLH-floating operations are performed. Emerging slithers, identified by an emerging gap are stored in the *slither list*. If a closing operation becomes possible, it is performed, the slither list is deleted and stage 1 is repeated from scratch.

Stage 2. An opening operation is performed followed by floating operations. If $n \geq 2$ slithers are considered, then stage 2 is repeated by performing a different opening operation until all possible opening operations are considered.

Stage 3. After one opening operation is performed, we add the obtained slither to the slither list and then attempt floating transformations that close a gap without retreating to already listed slithers. If no floating transformation that closes a gap is available, then repeat the above by performing another opening operation and then floating operation that closes a gap.
Stage 0. Only SLH-closing operations are performed until no more closing operation is possible

Stage 1. SLH-floating operations are performed. Emerging slithers, identified by an emerging gap are stored in the slither list. If a closing operation becomes possible, it is performed, the slither list is deleted and stage 1 is repeated from scratch.

Stage 2. An opening operation is performed followed by floating operations. If $n^2$ slithers are considered, then stage 2 is repeated by performing a different opening operation until all possible opening operations are considered.
Algorithm

- **Stage 0.** Only SLH-closing operations are performed until no more closing operation is possible.
- **Stage 1.** SLH-floating operations are performed. Emerging slithers, identified by an emerging gap are stored in the *slither list*. If a closing operation becomes possible, it is performed, the slither list is deleted and stage 1 is repeated from scratch.
- **Stage 2.** An opening operation is performed followed by floating operations. If \( n^2 \) slithers are considered, then stage 2 is repeated by performing a different opening operation until all possible opening operations are considered.
- **Stage 3.** After one opening operation is performed, we add the obtained slither to the slither list and then attempt floating transformations that close a gap without retreating to already listed slithers. If no floating transformation that closes a gap is available, then repeat the above by performing another opening operation and then floating operation that closes a gap.
We define the distance between current slither $S_G$ and hamiltonian slither $H_G$ as

$$\text{dist}(S_G, H_G) = n - k,$$

where $k$ is the number of common edges of $C$ and $H$.

The difference between $S_G$ and $H_G$ is defined as

$$\Delta(S_G, H_G) = \frac{g(S_G)}{3} + \text{dist}(H_G, S_G),$$

where $g(S_G)$ is the number of gaps in $C$.

Knowledge of $H$ allows us to apply some $\gamma$ or some $\kappa$ or a composition $\gamma \circ \kappa$ that reduces $\Delta(\cdot, H_G)$ every time the operation is performed.
Illustration of $\Delta -$ reducing operations

$C$ contains at least one gap $(x, y)$. 
In situations (i) and (iii) we choose $\gamma(y, x, a)$ that results in the slither $S'_G$, such that $\Delta(S'_G, H_G) \leq \Delta(S_G, H_G) - 1$, because the distance $\text{dist}(S'_G, H_G) \leq \text{dist}(S_G, H_G) - 1$.

Common edge $(x, a)$ joins the slither while foreign to $H$ snake $(a, b)$ becomes a ladder) and the number of gaps in $S'_G$ does not increase compared to $S_G$ (it could decrease, though if, for example, the edge $(y, b) \in E(G)$).

In situation (ii) there are three possibilities:

1. $(b, r) \in H$ We apply $\kappa(x, a, b, c)$ transform. The number of gaps will not increase by no more than 1. New slither contains new snakes $(x, a)$ and $(b, c)$ which are common with $H$ but might not contain $(c, f)$ that is a former snake.

2. $(b, d) \in H$. We apply $\gamma(e, d, b) \circ \kappa(y, x, a, b, c)$ transform. The number of gaps can not increase by more than 1. The new slither contains former ladders $(x, a)$ and $(b, c)$, might not contain $(c, f)$ that was a snake. Former snake $(b, d) \notin H$.

3. $(b, s) \in H$. In this case $\kappa(y, x, a, b, c)$ or $\gamma(e, d, b) \circ \kappa(y, x, a, b, c)$ decreases $\Delta$. 

Flinders Hamiltonian Cycle Project - http://fhcp.edu.au

Snakes and Ladders Heuristic
$S_G$ is some other Hamiltonian slither (a slither, corresponding to another Hamiltonian cycle):
If \( g(S_G) = 0 \), i.e. that \( S_G \) corresponds to a Hamiltonian cycle, different from \( H \). If \( H \) contains a ladder \((x, a)\) then, there is vertex \( y \), next to \( x \) such that the snake \((x, y)\) is not in \( H \). Treat \((x, y)\) as a gap, and apply those transformations as above in the corresponding situation to improve the dist function by at least 1 in every case. We may create at most 2 gaps if \( \pi \) or a composition of \( \pi \circ \gamma \) is applied. Overall improvement for the \( \Delta \) function will be at least \( \frac{1}{3} \), compared to \( \frac{2}{3} \), when \((x, y)\) is a genuine gap.
Flood (1956) discovered that 2-exchanges can improve the tours.

This was generalised later to 3-opt and k-opt.

A k-opt transformation exchanges a set of snakes \( X = \{x_1, x_2, \ldots, x_k\} \) by a set of ladders \( Y = \{y_1, y_2, \ldots, y_k\} \) to obtain a new tour such that the new tour is better than the previous.

However, since they had to pre-program every transformation, program would get stuck whenever the special structures were not present in a graph.

Lin and Kernighan (1973) invented a method to avoid pre-programming all transformations. The program could search for a plausible sequential exchanges and determine what is best to perform.
Example of $k$–opt sequential transformation:
Example of $k$–opt sequential transformation:
Example of $k$–opt sequential transformation:
Example of $k$-opt sequential transformation:
Example of $k$–opt sequential transformation:
Example of $k$–opt sequential transformation:
Example of $k$–opt sequential transformation:
Example of $k$-opt non-sequential transformation:
Example of $k$–opt non-sequential transformation:
Example of \( k \)-opt non-sequential transformation:
Example of HCP for a graph that requires a non-sequential exchange:

However, the following SLH transformation can find the Hamiltonian cycle

\[ T = \gamma(v_{13}, v_9, v_{15}) \circ \gamma(v_9, v_6, v_{14}) \circ \kappa(v_5, v_8, v_7, v_{13}) \circ \gamma(v_{15}, v_{10}, v_{11}) \circ \gamma(v_5, v_2, v_{12}) \circ \kappa(v_1, v_4, v_3, v_{11}). \]
SLH vs. $k$–opt: differences

SLH differs from $k$–opt algorithms in three important ways.

(1) The $k$-opt heuristics rely heavily on repeated randomisation techniques to obtain an optimal tour (Hamiltonian cycle), while SLH runs on any prescribed starting orientation in a deterministic fashion.

(2) All transformations of SLH are expressible as compositions of a sequence of two simple generator transformations: $\gamma$ and $\kappa$. Some of these transformations are not allowed, or are not generally used, in $k$–opt algorithms.

(3) The $k$-opt algorithms update the tour whenever an “improvement” (in decreasing the number of gaps) is possible, while SLH allows floating and opening transformations that may result in either no “improvement” or a “sacrifice”, respectively.
### SLH comparative performance

<table>
<thead>
<tr>
<th>Graphs</th>
<th>Concorde (Time/sec)</th>
<th>LKH (Time/sec, Success (%))</th>
<th>SLH (Time/sec, Stage)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALB 1000</td>
<td>4.95</td>
<td>0.0, 100</td>
<td>0.1, 1</td>
</tr>
<tr>
<td>ALB 2000</td>
<td>7.30</td>
<td>0.0, 100</td>
<td>0.8, 1</td>
</tr>
<tr>
<td>ALB 3000a</td>
<td>9.56</td>
<td>0.0, 100</td>
<td>3.44, 1</td>
</tr>
<tr>
<td>ALB 3000b</td>
<td>9.94</td>
<td>0.0, 100</td>
<td>3.64, 1</td>
</tr>
<tr>
<td>ALB 3000c</td>
<td>9.95</td>
<td>0.0, 100</td>
<td>4.31, 1</td>
</tr>
<tr>
<td>ALB 3000d</td>
<td>10.14</td>
<td>0.0, 100</td>
<td>4.03, 1</td>
</tr>
<tr>
<td>ALB 3000e</td>
<td>10.44</td>
<td>0.0, 100</td>
<td>3.29, 1</td>
</tr>
<tr>
<td>ALB 4000</td>
<td>13.45</td>
<td>0.0, 100</td>
<td>13.89, 1</td>
</tr>
<tr>
<td>ALB 5000</td>
<td>17.24</td>
<td>0.0, 100</td>
<td>14.12, 1</td>
</tr>
</tbody>
</table>

**Table:** Comparative performance of SLH.
<table>
<thead>
<tr>
<th>Graphs</th>
<th>Concorde</th>
<th>LKH</th>
<th>SLH</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Time (sec)</td>
<td>Time (sec)</td>
<td>Success (%)</td>
</tr>
<tr>
<td>GP39-2</td>
<td>34.38</td>
<td>0.0</td>
<td>99.9</td>
</tr>
<tr>
<td>GP45-2</td>
<td>50.91</td>
<td>0.0</td>
<td>98.6</td>
</tr>
<tr>
<td>GP51-2</td>
<td>50.09</td>
<td>0.0</td>
<td>90.0</td>
</tr>
<tr>
<td>GP63-2</td>
<td>737.88</td>
<td>0.1</td>
<td>32.1</td>
</tr>
<tr>
<td>GP123-2</td>
<td>Fail</td>
<td>0.4</td>
<td>00.4</td>
</tr>
<tr>
<td>GP243-2</td>
<td>Time Fail</td>
<td>1.5</td>
<td>00.0</td>
</tr>
</tbody>
</table>

**Table:** Comparative performance of SLH.
Conclusions

- Numerical experiments confirm that SLH is successful in finding Hamiltonian cycles in graphs possessing very few Hamiltonian cycles. This included graphs where other, benchmark, solvers failed to find such cycles.
- No need for randomisation techniques, for example randomising the assignment of the graph, that is a common feature in many contemporary HCP heuristics.
- Balancing the use of $\gamma$ and $\kappa$ generators overcomes some of the difficulties that other heuristics try to overcome by randomisation.
The current version of SLH is available on the web on

http://fhcp.edu.au/slhweb

where users are invited to solve their HCP problems, online, by means of SLH.