Permutation and Relabeling

- Permutation - change locations irregardless of values
  
  \[ P_a(P_b(S)) \neq P_b(P_a(S)) \]

- Relabeling - change values irregardless of location
  
  \[ R_a(R_b(S)) \neq R_b(R_a(S)) \]

- But, these can be mixed:
  
  \[ R_a(P_b(S)) = P_b(R_a(S)) \]

Duality

- For every state in many puzzles there are dual states as well

Example

- Sliding Tile Puzzle
- Pancake Puzzle
Duality Requirements

- State is represented by a permutation of constants
- Operators are location-based permutations
  - eg swap two locations; not tile specific
- Operators are invertible
  - Inverse operators have same time cost

Duality -- Caveat

- Duality requires location-based operators
  - Sliding-tile puzzle doesn’t quite work
- 0 1 2 | 3 4 5 | 6 7 8
  - Apply right, down, down
- 1 4 2 | 3 7 5 | 6 0 8
  - Inverse operators can’t be applied to start state
  - Just apply dual when blank in corner

Duality Uses

- Pattern Database lookup
  - Equivalent to asking the cost of moving tiles in PDB from their original positions inside the pattern to the goal location
- Example
  - Sliding Tile Puzzle
  - Pancake Puzzle

Duality - Consistency

- Do dual lookups result in consistent heuristics?
- Given two states $s_1$ and $s_2$, if $s_1$ is adjacent to $s_2$, is $\text{dual}(s_1)$ adjacent to $\text{dual}(s_2)$?
  - Consider the optimal path from the start to $s_1$ plus the action to get to $s_2$
  - The path (actions) to $\text{dual}(s_1)$ is identical to $\text{dual}(s_2)$ except the path to $\text{dual}(s_2)$ is preceded by the extra action
  - $\text{dual}(s_1)$ and $\text{dual}(s_2)$ aren’t necessarily adjacent
**DIDA***

- Using duality for search
  - Can search from regular or dual of a state
- Algorithm is identical to IDA* BUT:
  - Each time we reach a state, choose whether to search from dual or regular state

**DIDA* Notes**

- Why shouldn’t we switch all the time?
  - Branching factor increases
- Is the dual lookup consistent?
  - No. Improves search!

**Results**

<table>
<thead>
<tr>
<th>Time</th>
<th>Lookups</th>
<th>Type</th>
<th>Nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td>28.18 s</td>
<td>1</td>
<td>Regular</td>
<td>90,930,662</td>
</tr>
<tr>
<td>11.60 s</td>
<td>4</td>
<td>Regular</td>
<td>10,574,180</td>
</tr>
<tr>
<td>7.85 s</td>
<td>2</td>
<td>Regular</td>
<td>13,380,154</td>
</tr>
<tr>
<td>7.38 s</td>
<td>1</td>
<td>Dual*</td>
<td>19,653,386</td>
</tr>
<tr>
<td>3.30 s</td>
<td>1</td>
<td>Random*</td>
<td>9,652,138</td>
</tr>
<tr>
<td>3.24 s</td>
<td>1</td>
<td>Dual + BPMX*</td>
<td>8,315,116</td>
</tr>
<tr>
<td>1.25 s</td>
<td>1</td>
<td>Random + BPMX*</td>
<td>3,929,138</td>
</tr>
<tr>
<td>1.20 s</td>
<td>4</td>
<td>Random + BPMX*</td>
<td>1,042,451</td>
</tr>
<tr>
<td>1.14 s</td>
<td>2</td>
<td>Random + BPMX*</td>
<td>1,902,730</td>
</tr>
</tbody>
</table>

**Bidirectional Search**

- If we did a bidirectional search with BFS, we could reduce the effective depth by 2
  - As long as we use a hash table to look up states to see if the frontiers have met
  - O(b^{d/2})
- Doesn’t work as well with A* as frontiers might not meet
  - Could perform double the work
Bidirectional Search

- Can we generalize bidirectional search into a single frontier?
  - SFBDS
  - Each node in the tree has the start/goal explicitly stated
  - Can expand either side

When would we use SFBDS?

- If branching factor is fixed at each depth:
  - Only save the maximum branching factor
- If there are lots of duplicates?
  - O(N^2) possible tasks, will perform poorly
- If the heuristic (thus the branching factor) is non-uniform in two parts of the state space

Regular tree versus SFBDS tree

SFBDS is a generalization of DIDA*

- Searching from S to G
- DIDA* finds the dual and then applies the operator (permutation) \( P_o \) before looking up the heuristic to the goal
  - \( S' = P_{s\rightarrow g}(G) \) [Find the dual]
  - \( P_o(P_{s\rightarrow g}(G)) \) [Apply the next operator, lookup \( h(P_o(P_{s\rightarrow g}(G)), G) \)]
- SFBDS is searching from G to S with remapping:
  - \( R_{s\rightarrow g}(S) = G \) [for the heuristic lookup]
  - \( P_o(G) \) [Apply the next operator, lookup is \( h(R_{s\rightarrow g}(P_o(G)), G) \)]
  - Also note \( P_{s\rightarrow g}(S) = G \). Proof that heuristic lookups are identical:
    - \( R_{s\rightarrow g}(P_o(G)) = P_o(R_{s\rightarrow g}(G)) = P_o(R_{s\rightarrow g}(P_{s\rightarrow g}(S))) = P_o(P_{s\rightarrow g}(R_{s\rightarrow g}(S))) = P_o(P_{s\rightarrow g}(G)) \) [The same heuristic looked up my DIDA*]