Abstract
Beam search is an optimization of the best-first graph search algorithm. Its improvement, beam-stack search, is a complete search algorithm that is guaranteed to find an optimal solution. This paper gives an overview of beam-stack search, and a memory-efficient version of beam-stack search, called divide-and-conquer beam-stack search. The advantages of beam-stack search are demonstrated by presenting some computational results.

Beam search
Beam search is a heuristic graph search algorithm that explores a graph by expanding the most promising nodes of each layer using breadth-first order, [Wikipedia a] and discarding the rest [Zhou and Hansen, 2005].

The time and memory complexity of beam search is linear - \( wd \), where \( w \) is the number of promising nodes (called beam width) and \( d \) is the depth of the search [Zhou and Hansen, 2005].

The main drawback of beam search is its incompleteness – since beam search is inadmissible, there is no guarantee that it will find an optimal solution, or any solution at all.

Beam-stack search
Beam-stack search is an improvement of beam search, combined with backtracking [Wikipedia b]. Similarly to beam search, it expands nodes in breadth-first order, but in addition it uses a technique called admissible pruning – the use of upper and lower bounds to prune the search space. The lower-bound estimate of the cost of an optimal path through a node is described by a function \( f(n) = g(n) + h(n) \), where \( g(n) \) is the cost of best path from the start node to node \( n \), and \( h(n) \) is an admissible heuristic that never overestimates the remaining cost from node \( n \) to goal node [Zhou and Hansen, 2005]. The upper-bound estimate \( U \) can be provided by the user.

To reimburse for inadmissible node discarding in beam search, beam-stack search can backtrack to a previously-explored layer, using a data structure called beam stack [Zhou and Hansen, 2005]. Beam stack contains one item for each layer of the breadth-first search graph (except the deepest layer where it is not necessary to backtrack to).

Beam stack keeps track of which nodes have been considered so far by regarding that the nodes in a layer can be sorted using their unique \( f \)-cost. An item of the beam stack describes a range of \( f \)-costs which indicates that only nodes with an \( f \)-cost in the range of \( [f_{\min}; f_{\max}] \) are stored in that layer [Zhou and Hansen, 2005].

When beam-stack search backtracks to a layer, it uses the current \( f_{\max} \) as the new \( f_{\min} \) and the upper bound \( U \) as the new \( f_{\max} \).

Beam-stack search finishes when all layers are backtracking complete (the beam stack is empty). Beam-stack search is an anytime algorithm – it finds an initial solution relatively quickly, but continues to search for improved solutions until finding the optimal solution. Each time beam-stack search finds an improved solution, it updates its upper bound [Zhou and Hansen, 2005].

Memory efficiency
The memory complexity of both beam search and beam-stack search is linear – \( dw \), where \( d \) is the depth of the search and \( w \) is the beam width. This means, for a deep search, the beam width \( w \) must be small, so the stored nodes in all layers could fit in a fixed amount of memory.

There is an implementation of beam search called divide-and-conquer beam search (DCBS) that reduces memory requirements. The memory complexity of DCBS is \( 4w \) – meaning that it is independent of the depth of the search [Zhou and Hansen, 2005], so it allows much larger beam widths.

This section gives an overview of divide-and-conquer beam search and shows how this technique can be used to improve the memory efficiency of beam-stack search.
Divide-and-conquer beam search

The main idea behind divide-and-conquer beam search is the fact that it is not necessary to store all explored nodes in memory to detect if some newly-generated node is a duplicate of an already-explored node [Zhou and Hansen, 2005].

In divide-and-conquer beam search, each node past the midpoint stores a pointer to an intermediate node (called a relay node) that is kept in memory; nodes before the midpoint store a pointer to the start node. All relay nodes are stored in the same layer, called the relay layer of the search graph. It could be the middle layer, but it is usually more efficient when it is \( \frac{3}{4} \) layer, since that one is usually smaller. As a result, divide-and-conquer beam search stores four layers – the currently-expanding layer, its successor layer, its previous layer, and the relay layer [Zhou and Hansen, 2005].

The memory complexity of divide-and-conquer beam search is constant instead of linear \((4w \text{ instead of } dw)\) [Zhou and Hansen, 2005].

Divide-and-conquer beam-stack search

Divide-and-conquer technique can be combined with beam-stack search same ways as with beam search, creating an algorithm called divide-and-conquer beam-stack search (DCBSS). But combining divide-and-conquer technique with backtracking introduces a problem – since divide-and-conquer beam-stack search only keeps four layers of the search graph in memory, the layer to which it backtracks may not be in memory [Zhou and Hansen, 2005]. In this case, the algorithm must recover the missing layer.

Using external memory

There is a method for reducing the time cost of the DCBSS algorithm considerably. For that, an external-memory version of DCBSS can be created – when DCBSS backtracks to a missing layer, it copies the previously-explored layers to disk and copies them back to internal memory as needed [Zhou and Hansen, 2005].

Computational results

Computational results show that divide-and-conquer beam-stack search is able to find optimal solutions for problems that cannot be solved optimally by the previous best algorithm, divide-and-conquer breadth-first branch-and-bound [Zhou and Hansen, 2005].

Table 1 shows how much the time cost of divide-and-conquer beam-stack search can be reduced when using external memory [Zhou and Hansen, 2005].

<table>
<thead>
<tr>
<th>Problem</th>
<th>Len</th>
<th>Stored in RAM</th>
<th>No external memory</th>
<th>External memory</th>
<th>Stored on disk</th>
<th>Exp</th>
<th>Secs</th>
</tr>
</thead>
<tbody>
<tr>
<td>logistics-6</td>
<td>25</td>
<td>48.253</td>
<td>1.608.139</td>
<td>16.4</td>
<td>127.851</td>
<td>770.187</td>
<td>13.8</td>
</tr>
<tr>
<td>blocks-14</td>
<td>38</td>
<td>39.823</td>
<td>593.377</td>
<td>17.3</td>
<td>100.328</td>
<td>365.199</td>
<td>7.4</td>
</tr>
<tr>
<td>gripper-6</td>
<td>41</td>
<td>296.005</td>
<td>10.179.757</td>
<td>448.9</td>
<td>1565.800</td>
<td>775.639.416</td>
<td>6907.7</td>
</tr>
<tr>
<td>satellite-5</td>
<td>20</td>
<td>796.049</td>
<td>12.343.501</td>
<td>232.7</td>
<td>2.213.275</td>
<td>1265.645.271</td>
<td>3.8730</td>
</tr>
<tr>
<td>elevator-11</td>
<td>37</td>
<td>488.777</td>
<td>8.995.165</td>
<td>306.9</td>
<td>1.922.041</td>
<td>6.635.998</td>
<td>233.3</td>
</tr>
<tr>
<td>depots-3</td>
<td>27</td>
<td>1.592.785</td>
<td>62.985.231</td>
<td>4767.1</td>
<td>3.640.367</td>
<td>30.426.421</td>
<td>2.3129</td>
</tr>
<tr>
<td>driverlog-10</td>
<td>17</td>
<td>1.127.758</td>
<td>-</td>
<td>-</td>
<td>1.922.041</td>
<td>6.635.998</td>
<td>233.3</td>
</tr>
<tr>
<td>freecell-4</td>
<td>27</td>
<td>1.565.164</td>
<td>-</td>
<td>-</td>
<td>3.640.367</td>
<td>30.426.421</td>
<td>2.3129</td>
</tr>
</tbody>
</table>

Table 1: Comparison of divide-and-conquer beam-stack search with and without using external memory, under the same internal memory constraints [Zhou and Hansen, 2005]

Conclusion

In this paper, an improvement of beam search, called beam-stack search, was described. It was shown how the combining of divide-and-conquer technique with beam-stack search, although producing some problems, can be implemented. By using external memory version of divide-and-conquer beam-stack search the memory efficiency can be increased notably, as was also illustrated by computational results.

References


http://en.wikipedia.org/wiki/Beam_search

http://en.wikipedia.org/wiki/Beam_stack_search