Effective use of memory in linear space best first search

Marko Robnik-Šikonja
University of Ljubljana,
Faculty of Electrical Engineering and Computer Science,
Tržaška 25, 1001 Ljubljana, Slovenia
e-mail: Marko.Robnik@fri.uni-lj.si

Abstract

Classical best-first search algorithms like A* require space exponential in length of the solution path. To overcome this limitation algorithms like IDA* and RBFS regenerate parts of problem space trading space for time. These algorithms are probably too modest using only a tiny amount of memory today's computers can offer. We present a novel, RBFS based algorithm which uses all available memory. This is achieved by keeping the generated nodes in memory and pruning some of them away when the memory runs out.

We describe three pruning methods and experiments confirming the advantage of our approach.

1 Introduction

Although heuristic search is well-established area of research, new findings have great practical significance because of large number of applications and because search underlies many complex systems.

In this paper we adopt a standard framework of heuristic search in which we try to find a sequence of steps from a given initial state to any goal state. The search is guided by a heuristic function $f(n) = g(n) + h(n)$, where $g(n)$ is the cost of state n and $h(n)$ is the heuristic estimation of the lowest cost of any path from state n to the goal state. We will assume an admissible heuristic function $h(n)$, which never overestimates the actual cost of the path. Presented algorithms are guaranteed to find an optimal solution in this case [3].

Well known disadvantage of classical best-first search algorithms such as A* [3], is their exponential memory requirement which often prevents them from finding optimal solutions to many problems.

A* maintains OPEN and CLOSED lists of generated nodes, keeping them in memory which results in space requirement $O(b^d)$, where d is the length of the solution, and b is the average branching factor. Additional secondary storage does not help as the nodes are accessed randomly.

IDA* (Iterative Deepening A*) [6] was the first algorithm which addressed the memory complexity of best-first search. It requires only linear space: $O(d)$. IDA* iteratively deepens search horizon until it reaches an optimal solution. In one iteration it explores nodes in depth-first order up to the current depth (cost). Parts of the problem space closer to the root of the search are regenerated in each iteration. This is how IDA* trades space for

time.

Korf's RBFS (recursive best first search) algorithm [7] is further improvement of IDA*, using slightly more memory (in order of $O(b \cdot d)$) but generating asymptotically less nodes and having also other desirable features regarding nonmonotonic cost function. RBFS keeps essential information of already generated problem space and can regenerate nodes more cleverly.

IDA* and RBFS demand memory in order of $O(d)$ and $O(b \cdot d)$, respectively. For most problems this is a tiny amount of memory available on today's computers.

We present an extension of RBFS, which allows the use of all available memory. When the improved algorithm fills the available space the question arises how to proceed? Our answer is presented in the form of pruning strategies. By controlling which part of the generated search space we keep and which we prune away, we can cache most promising parts of the search space. This prevents their frequent regeneration and saves time.

We have performed a series of experiments on the sliding-tile puzzle and travelling salesman problem to compare the improved variants and the original algorithm. The results show clear advantage of our approach.

We first present the essentials about RBFS, Section 3 contains description of the improvements and Section 4 deals with the experiments. The last two sections discuss related work and give conclusions.

2 RBFS

Our work extends Korf's RBFS (Recursive Best First Search) algorithm [7], therefore we briefly present the way it works.

RBFS explores the search space as it were a tree. An example of the search space with cost equal to depth is in Figure 1. Figure 2 shows the sequence of steps RBFS makes while exploring this tree. RBFS is called with two arguments: a node to expand and an upper bound on the cost of the node's subtree. It then explores the subtree below this node as long as it contains the frontier nodes whose costs do not exceed the upper bound. At this point it returns the minimum cost of the frontier nodes in the explored subtree. The initial call on the root sets the upper bound to infinity.

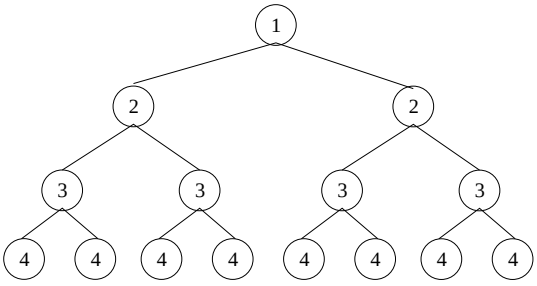


Figure 1: Example of search space with costs equal to depth.

The Figure 2a shows the call on the root and generation of its children. As both children are of equal quality, the left (the first one) is examined first, with the bound 2 i.e. minimum of its parent and its best sibling (Figure 2b). As both of its descendants exceed this bound, the algorithm backtracks and starts examining the right subtree, but it stores the cost of the minimum bound breaking node (3) in the left subtree (Figure 2c). Figures 2d and 2e show an expansion of the right subtree, while Figure 2f gives details when algorithm returns from the right subtree, updates its stored value to 4, and starts regenerating the left subtree. Since the left subtree has already been explored up to the cost of 3, it is now safe to move up to that bound in depth-first manner (Figure 2g) and proceed from there in a best-first manner again (Figures 2h and 2i).

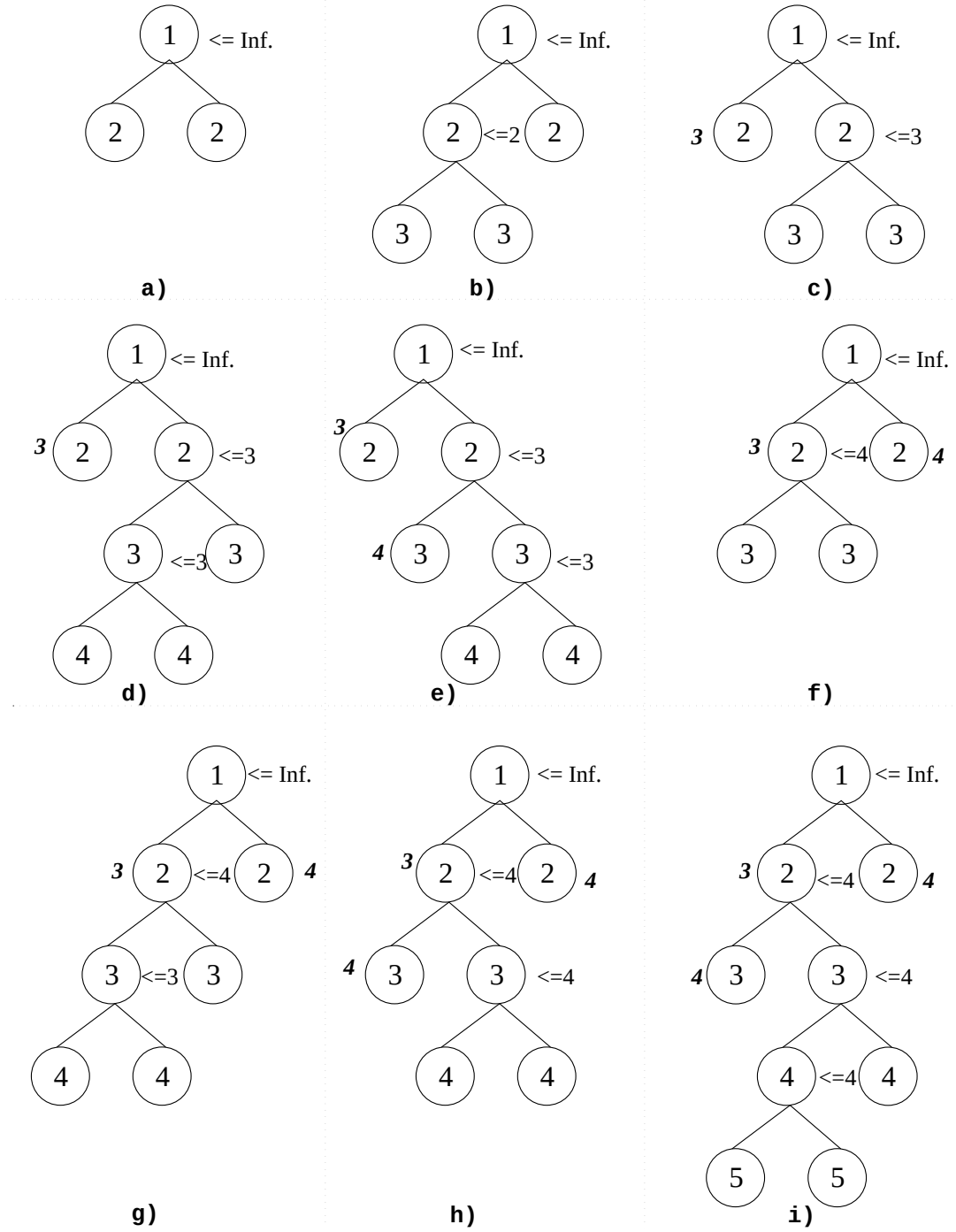


Figure 2: The sequence of steps RBFS makes when exploring a tree with costs equal to depth

To achieve the described behaviour, besides static heuristic estimate $f(n)$, RBFS maintains also backed-up value $F_b(n)$ for each node. $F_b(n)$ is initialized with $f(n)$ when the node is generated for the first time. When the algorithm explores the whole subtree of the node up to the given bound and returns, this value is updated to the minimum cost of the frontier nodes in this subtree, that is with the cost of the node which minimally exceeds the given bound. In this way the essential information on the quality of the subtree is preserved for possible subsequent regenerations. The updated value $F_b(n)$ of the node serves also as an indicator if the node has been already explored. If $F_b(n) > f(n)$ then the node n has been explored previously and the F_b values for the children of n can be set to the maximum of their parent updated value $F_b(n)$ and their respective static values f. In this way we are allowed to search already explored space in the depth-first manner, without unnecessary backtracking the best-first behaviour would probably require. Pseudo-code of the algorithm is in Figure 3, but consider only unmarked lines.

3 Using more memory with MRBFS

The comparison of the sequence of steps RBFS makes in Figures 2 with Figure 1, which shows the whole problem space, reveals that RBFS keeps in memory slightly more than the current search path. To be exact: it keeps all the nodes on the current search path and all their siblings. This explains the algorithm's space complexity $O(b \cdot d)$. In this paper we will call these nodes the skeleton or the backbone nodes.

For most problems RBFS is very modest what concerns space. For example, at sliding-tile puzzle (15 squares) where on average the solution length is 52 moves and branching factor is between 2 and 3, RBFS keeps in memory less than 150 nodes. Even with casual representation of nodes, the total amount of memory the algorithm needs is orders of magnitude less than today's computers can offer.

Knowing this, it is natural to try to change RBFS in a way to use more memory, to possibly use all the available memory in order to reduce the search time.

Let's imagine for a moment that we have adopted the strategy to use RBFS, but to keep all the generated nodes in memory eliminating the need to ever do any regeneration. Such strategy, apart from being impossible for most problems on real computers (as A*, RBFS would also fill all the available memory quickly), would save the time for regeneration of the nodes as the backed-up values would be stored and would guide the search towards the best frontier node while reexamining certain subtree.

To adopt the above scenario on a real computer we have to answer the question what to do when we fill up the memory. Our goal is to minimize the search time and with the good nodes in the memory the probability for the regeneration is decreasing. Now we can define our task: *exploit the memory well, but do not spend too much time for that.*

In the spirit of the above discussion we propose three different pruning strategies which are activated when the memory is full.

Prune all: we prune away from the tree all but skeleton nodes, leaving only the nodes RBFS would produce. This strategy does not use a lot of time for pruning, but we profit only with regenerations which would occur during one filling of the memory.

Prune the worst subtree: among the subtrees with the root on the skeleton we search for the worst one and prune it

away. With this strategy we spend some time to find the worst subtree, but the memory is used much better, as the majority of good nodes remain in the memory, so the probability to regenerate them is lower. If we are lucky, we may completely avoid regenerations.

Prune the worst node: among the nodes in the memory we select one with the worst value of F_b and prune it. It is obvious that this strategy spends quite some time with pruning, but the memory is used optimally.

To implement each of the above strategies we have to make the representation of nodes explicit, so that we can freely manipulate them in the memory (RBFS needs not worry about that as it stores the nodes on the stack with recursive calls).

For each of the strategies we will describe changes needed in RBFS and give some details about their respective implementations.

3.1 Prune all

Figure 3 contains the pseudo-code of RBFS with changes needed to adopt pruning. "Prune all" strategy contains non-marked lines and lines marked with *S*.

RBFS keeps descendants of a node sorted, with the best being in the first place, so the procedure *pruneSearchTree()* just moves down the skeleton following the first offspring and pruning the subtrees of its siblings, until it reaches an unexamined node.

3.2 Prune the worst subtree

Changes in RBFS on Figure 3 are marked with *S*.

In *pruneSearchTree()* we first collect all skeleton nodes except the ones on the current search path in a set of candidates, then we consequently prune the worst among the candidates until we have freed a certain proportion of memory. This is necessary if we wish to save some time, otherwise we would spend too much time collecting the pruning candidates.

3.3 Prune the worst node

Changes in RBFS are marked with *N* on Figure 3. In order to make this strategy useful we have to make some preparations in advance. It would take unacceptably large amount of time if we tried to find the worst node from the search tree. Therefore during the search we keep the track of all the interior non-skeleton nodes in a separate partially ordered tree (POT), which allows fast insertion and retrieval of the worst node (in $O(log(n))$ time).

Procedure *pruneSearchTree()* is quite simple: it deletes the worst node from POT and prunes its subtree from the memory and POT.

We only prune the interior nodes together with their subtrees. If we allowed pruning of a single leaf, we would also have to change the generation of children to enable the creation of a single child; the search algorithm would have to be changed to cope with this, as well.

When the search comes to a node with already generated descendants we have to delete it from the heap, since it has become a skeleton node and we do not want to cut the branch we are currently investigating. Also when the algorithm returns from a subtree we have to insert its root into the heap, as it is no longer on the skeleton.

Of course we need some additional memory for this strategy. Each node needs a pointer to its position in the heap and all the interior, non-skeleton nodes need their representation in the heap: the pointer to the node in memory. The size of the heap is small comparing to the memory used, because we keep only the interior nodes in it.

```
   function ?RBFS(Node:searchNode; B:bound):value;
   {
    if Node.f > B  then
      return Node.f;
    if goal(Node)  then
    { heureka;  exit algorithm }
SN  if Node.noChildren = 0  then
SN {  // Node is not expanded
      createChildren(Node);
SN    NodesInMemory:=NodesInMemory+Node.noChildren;
SN    if NodesInMemory > MaxNodesInMemory  then
SN     pruneSearchTree();
SN }
N   else POT.delete(Node);
    if Node.noChildren = 0  then
      return ∞;
    for i := 1 to Node.noChildren  do
      if Node.f < Node.Fb  then
        Child[i].Fb := max(Node.Fb, Child[i].f)
      else Child[i].Fb:=Child[i].f ;
    sort children in ascending order of Fb
    while Child[1].Fb ≤ B and Child[1].Fb < ∞  do
    {
      if Node.noChildren = 1  then FbSibling := ∞
      else FbSibling := Child[2].Fb ;
      Child[1].Fb := ?RBFS(Child[1], min(B,
                                      FbSibling));
      insert Child[1] into children table in
                      increasing order of Fb
N     POT.insert(Child[1]);
    }
    return Child[1].Fb;
   }
```

Figure 3: Pseudo-code of variants of RBFS algorithm, where lines without marks belong to original algorithm, additional lines marked with S show changes needed for "all" and "subtree" strategies, and additional lines marked with N show changes needed for "node" strategy.

3.4 Features of MRBFS

The improved MRBFS (Memory-aware Recursive Best First Search) searches the tree in the best-first order. The proof for it is basically the same as for the original RBFS which can be found in [7]. In general the variants do not find the same solution. The reason is that different nodes are stored in the memory. The stored nodes guide the search in the direction of the best frontier node, while regeneration goes in the depth-first manner.

4 Experiments

We have conducted a series of experiments on the sliding-tile puzzle and the traveling salesman problem to verify the contribution of our approach. We have compared behaviours of the original RBFS and the three improvements: prune all, prune worst subtree and prune worst node, each variant with different amounts of available memory.

The memory was always measured in the number of available nodes. This gives a slight advantage to prune worst node strategy which, in our implementation, uses approximately 10% more memory per node than the other two strategies. This difference was neglected in our experiments. The results confirm that the conclusions remain unchanged when taking the difference into account.

Although we can pass a proportion of freed memory in the prune worst subtree strategy as a parameter, in described experiments it has always been set to 10%.

All the experiments were conducted on the Pentium-90 machine.

4.1 Sliding-tile puzzle

With sliding-tile puzzle we used Manhattan distance as an admissible heuristic: the sum over all tiles of the number of grid units each tile is displaced from its goal position. This heuristic is actually quite imprecise giving the same estimate to a large number of positions.

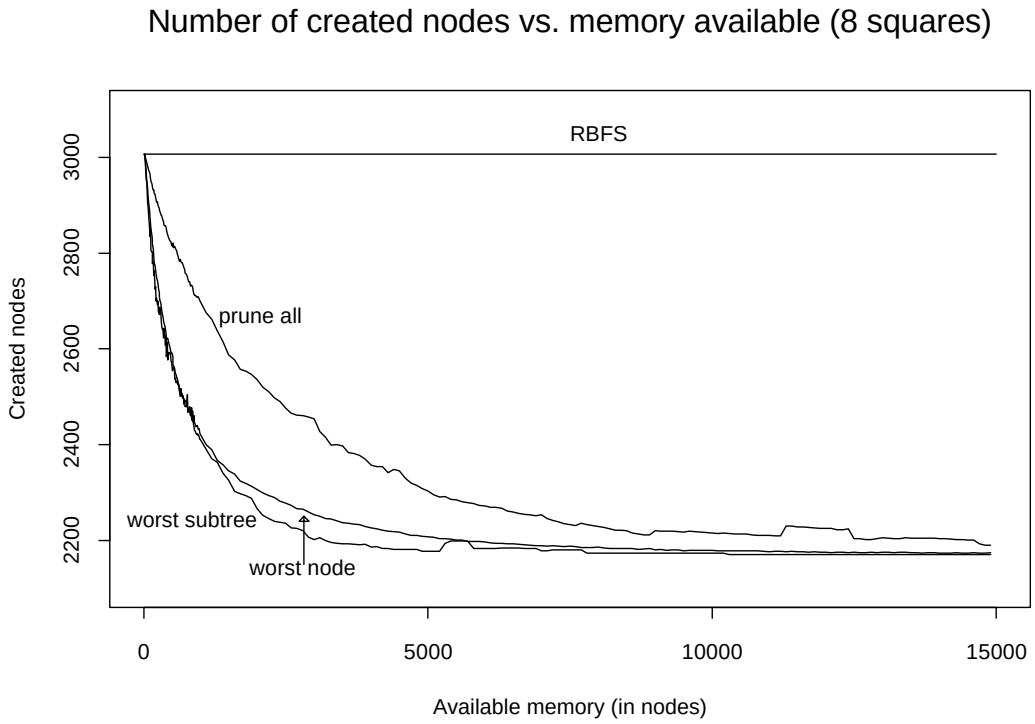
Figure 4 shows the relation between the number of created nodes (search time) and available memory in Eight Puzzle. Each point on the graph is the average of 1000 positions, altogether there are 240 computed points. Since Eight Puzzle is a small problem we were able to test it with enough memory that all the nodes fit into the memory.

If you note the scale on both graphs, you can see that while memory-aware versions generate approximately one third less nodes than RBFS, they save only one sixth of the time. The difference comes from larger time per node generation for memory-aware versions. Also we can see on the right side of the time graph that "worst node" has slightly larger overhead than "worst subtree" strategy. The latter would be chosen as a preferred strategy for Eight Puzzle, while "prune all" is obviously inappropriate also because of its unwanted behaviour in the left part of the time graph - its overhead with small amounts of memory is too high.

The roughness of the "created nodes" graph can be explained with different solutions found with different amounts of memory. Even larger roughness of time graph has additional cause in interactions with operating system which cannot be totally expelled from measurements (eg. cache misses).

We could not afford such extensive testing for Fifteen Puzzle, therefore we have tested with 50000 nodes of memory and averaged the results of 25 trials. Table 1 shows the results.

We can see that prune all and worst subtree do save some node regenerations (slightly less than one per mille), but not enough, to compensate larger time per node generation. Worst node pruning fails completely, the reason for this lies in large number of positions



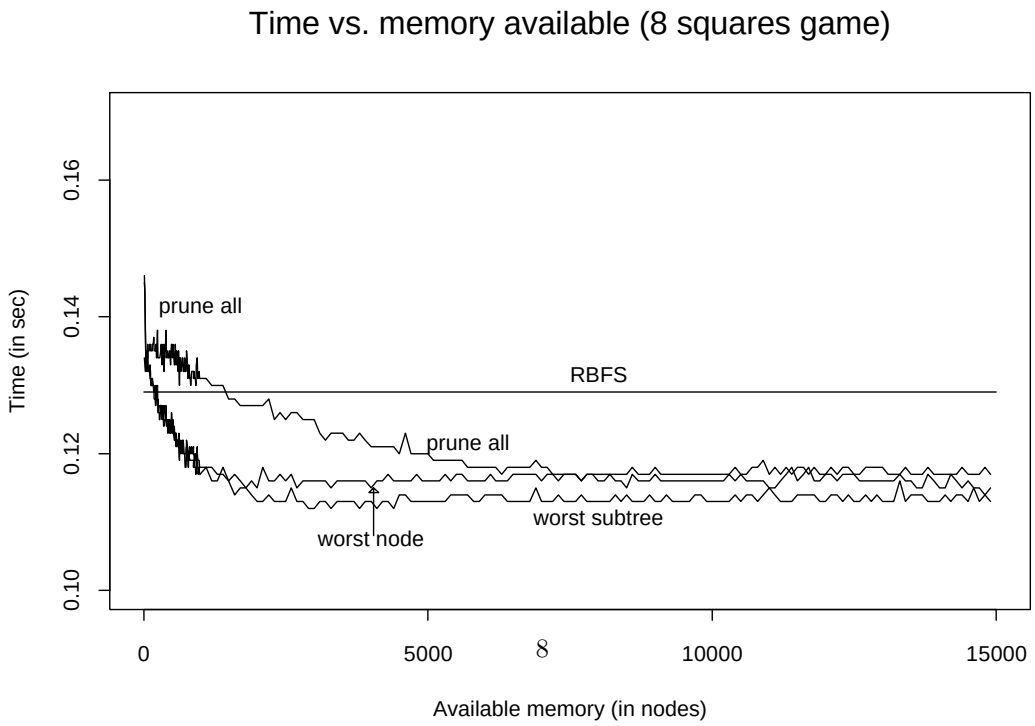


Figure 4: Created nodes and time versus available memory in Eight Puzzle.

Table 1: Optimal results for Fifteen Puzzle.

	RBFS	all	subtree	node
created	129329794	129313955	129211267	146680498
time	10814	11610	11518	12804

having the same estimate. During the experiment we could observe that almost all the time all 50000 nodes in the memory were having the same F_b value, which resulted in the algorithm blindly choosing the node to delete from the heap.

On the sliding-tiles puzzle RBFS is very hard to beat, as its node regeneration overhead (defined as the quotient of the number of regenerated nodes and newly generated nodes) is merely 0.206 for Fifteen Puzzle and 0.383 for Eight Puzzle, which leaves very little room for any improvements.

If we omit the demand for the optimal solution and use weighted evaluation function $f(n) = g(n) + Wh(n)$ the improved variants get better chances, as the estimations are more dispersed. Table 2 shows the average of 100 trials with $W = 3$. We can see that the improved algorithms reduce the search time by 40% compared to RBFS, while the the solution length equally degrades: from 52 for optimal search to 79.

4.2 Traveling salesman problem

We have tested two variants of the travelling salesman problem (TSP): Euclidean and asymmetric. As a heuristic function we have used the minimum spanning tree of the cities not visited yet. Many nodes in the search tree have unique values, making the TSP quite different to sliding-tiles.

Figure 5 shows the relations between the search time and the available memory for the Euclidean TSP with 10 and 12 cities. The relation between the number of created nodes and memory show the same tendency (RBFS creates on average 35531 and 3280023 nodes for 10 and 12 cities respectively, while the worst node strategy with 8000 nodes of memory creates 987 and 11844 nodes respectively. The asymmetric TSP behaves similarly.

Each of altogether 240 computed points in each graph is average of 100 positions. Note that while the abscissæ are the same on both graphs, the ordinates differ for several orders of magnitude!

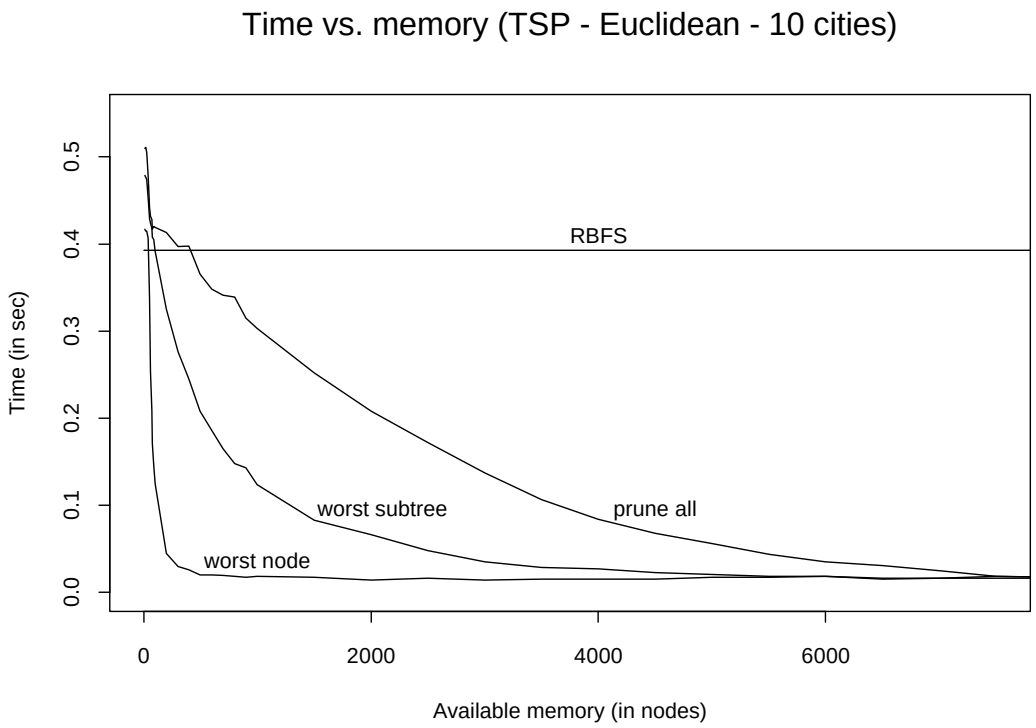
The clear winner among the compared approaches is the prune worst node strategy. A large number of distinct estimations gave this approach a chance to differentiate among the paths and cache the most promising. Even for small amounts of memory this approach saved several orders of magnitude in both, the number of generated nodes and time.

5 Related work

Korf's RBFS [7] inspired us for this study. It also gives some proofs and explanations which we only briefly mention here.

The MREC algorithm [9] stores nodes in memory until it is almost full, then performs IDA* starting from stored nodes. MREC searches like A* while MRBFS has the properties of RBFS and also prunes the memory thereby reducing the probability for regeneration.

MA* [2] and SMA* [8] dynamically store the best generated nodes so far and prune



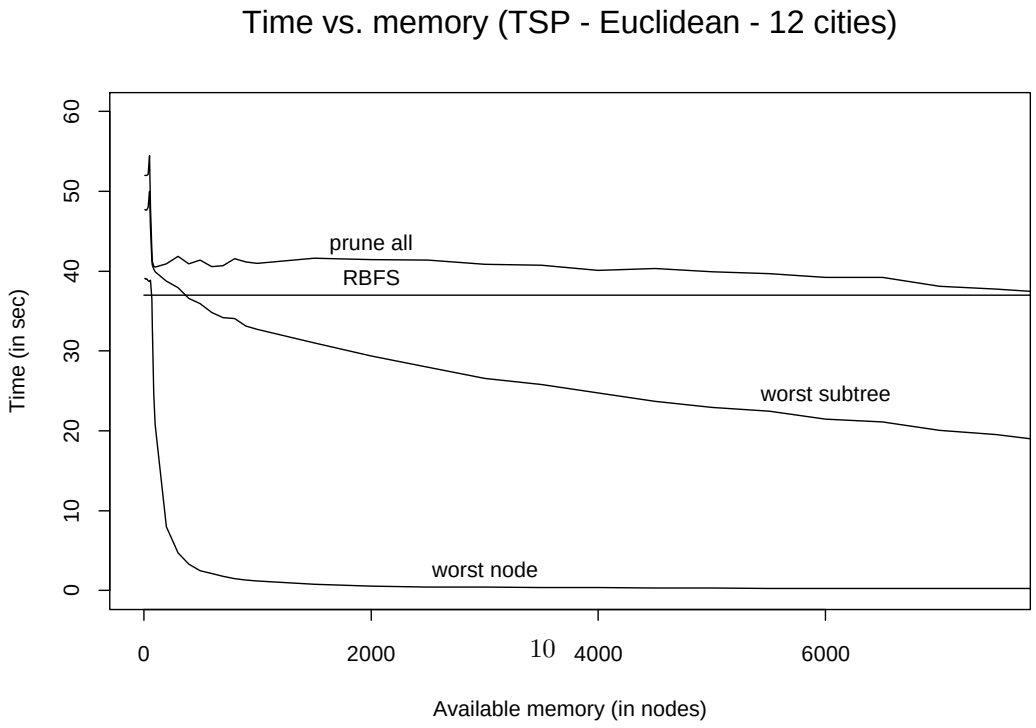


Figure 5: Time versus available memory in Euclidean traveling salesman problem (10 and 12 cities). Note the different scales!

Table 2: Results for Fifteen Puzzle with $W = 3$

	RBFS	all	subtree	node
created	173617	105242	98021	96506
time	22.3	14.37	13.47	13.46
length	79.11	79.67	79.73	79.67

away worst nodes when the memory runs out. Both algorithms, like A*, maintain OPEN and CLOSED lists, while (M)RBFS uses tree, which seems to have smaller overhead at updating nodes. Another overhead of MA* and SMA* is their demand for generation of a single successor.

Bratko's best-first search [1] searches in the same way as MRBFS and fills the memory with nodes, but it does not prune.

Kaindl et al. [4] propose the use of memory in bidirectional search. Because of its ability to cache the best nodes, MRBFS seems to fit nicely into their scheme. MRBFS would also fit into a number of suboptimal search schemes eg. [5].

6 Conclusion

Memory aware variants of RBFS algorithm keep all the nice properties of the original, but have additional power when there is enough memory available. If the savings in node regeneration are not big enough to compensate larger time per node generation, it is easy to automatically switch back to pure RBFS behaviour, as all the algorithms share common data representation.

Our experiments show that when the cost function produces many distinct values MRBFS is especially successful. There are many important problems with this property. In such cases the prune worst node strategy is a promising method. Its optimal use of available memory saves orders of magnitude of time even with relatively small amount of memory.

MRBFS caches generated nodes for further use. With different pruning strategies it enables flexible exploitation of available memory.

Acknowledgements

I would like to thank to Ivan Bratko who presented me the RBFS algorithm and to Igor Kononenko and Uroš Pompe who commented earlier drafts of the paper.

References

[1] Ivan Bratko, *Prolog Programming for Artificial Intelligence*, Addison-Wesley, Reading, Massachusetts, 1986.

[2] P.P Chakrabarti, S. Ghose, A. Acharya, and S.C. de Sarkar, 'Heuristic search in restricted memory', *Artificial Intelligence*, **41**(2), 197–221, (1989).

[3] P.E. Hart, N.J. Nilsson, and B. Raphael, 'A formal basis for the heuristic determination of minimum cost paths', *IEEE Transactions on Systems, Science and Cybernetics*, **4**(2), 100–107, (1968).

[4] Hermann Keindl, Gerhard Kainz, Angelika Leeb, and Harald Smetana, 'How to use limited memory in heuristic search', in *Proceedings IJCAI-95*, ed., Cris S. Mellish, pp. 236–242. Morgan Kaufmann, (1995).

[5] Hermann Keindl, Angelika Leeb, and Harald Smetana, 'Improvements on linear-space search algorithms', in *Proceedings ECAI-94*, pp. 155–159, (1994).

[6] Richard E. Korf, 'Depth-first iterative deepening: an optimal admissible tree search', *Artificial Intelligence*, **27**, 97–109, (1985).

[7] Richard E. Korf, 'Linear-space best-first search', *Artificial Intelligence*, **62**, 41–78, (1993).

[8] Stuart Russel, 'Efficient memory-bounded search methods', in *Proceedings ECAI-92*, ed., B. Neumann, pp. 1–5. John Wiley & Sons, (1992).

[9] Anup K. Sen and A. Bagchi, 'Fast recursive formulations for best-first search that allows controlled use of memory', in *Proceedings IJCAI-89*, pp. 297–302, (1989).