Chapter 4

The pn^2-search algorithm

One of the drawbacks of proof-number search (pn search) is that the whole search tree has to be stored in memory. Since computers are fast, the search tree grows quickly, causing the memory to be filled up completely. When the memory is full, the search process has to be terminated prematurely. Consequently, no solution will be found. For reducing memory usage, Allis *et al.* (1994) suggest two techniques which reduce the size of a generated search tree: the DeleteSolvedSubtrees technique and the DeleteLeastProving technique. The first technique removes all nodes which are solved. The technique is not very successful in searches which fail to determine the root value. The second technique removes parts of the tree least likely needed in the search. In this chapter we introduce a third technique which increases the information of the nodes in the tree in order to guide the search in a better way, thereby finding a solution more quickly. By this method we attempt to obtain more insight into the second problem statement of this thesis: which methods exist for best-first search to reduce the need for memory by increasing the search, thereby gaining more knowledge per node?

Section 4.1 introduces the pn^2-search algorithm. Details concerning this algorithm are discussed in Section 4.2. Section 4.3 presents the experiments. The results of the experiments are listed in Section 4.4. Section 4.5 states the conclusions.

4.1 Pn search with small memory: pn^2 search

Gaining more knowledge per node searched can be realized by using a better evaluation function at the leaves. One way of doing this is to use a *search process* at the leaves to obtain a more accurate evaluation. This method is used by other researchers as well. Berliner (1979) already used this idea in the B* algorithm, in which a shallow $\alpha\beta$ search evaluates the leaves. Pijls and De Bruin (1994) described the RSEARCH algorithm, in which certain leaves (the so-called *pseudo-terminals*) are evaluated by doing another RSEARCH. Recently, Baum and Smith (1997) reported on their Bayesian model of searching game trees: a two-stage Bayesian search is

performed in which an outer search is called by the inner search as its evaluation function. For the *pn^2-search* algorithm, introduced here, we also use this idea. It is briefly mentioned by Allis (1994), but so far no thorough research has been done on pn^2 search.

Pn^2 search is a search process consisting of two levels of pn search. The first-level search builds a tree in the same way as the standard pn-search algorithm for trees, as described in Section 3.1. However, the evaluation of the most-proving node is not performed by an evaluation function, but by a second-level pn search. The most-proving node of the first-level search tree acts as the root of the second-level search tree. The leaves in the second-level search are evaluated in the standard way, i.e., by an evaluation function returning one of the values *true*, *false*, and *unknown*. The leaf values are backed-up as usual leading to an evaluation of the most-proving node of the first-level pn search with more knowledge (acquired by using the second-level pn search) than in the standard way. After termination of the second-level pn search, the second-level tree is disposed of and the first-level search tree is updated using the new proof and disproof numbers of the most-proving node.

For pn^2 search the same tree-creation variants exist as in standard pn search: *immediate evaluation* and *delayed evaluation* (see Section 3.1). For the first-level search, the evaluation of a leaf takes much time, since it is a (pn-)search process itself. Therefore, it is efficient to use the *delayed-evaluation* variant for the first-level pn search. A limited set of experiments has shown that the delayed-evaluation variant indeed performs better for the first-level pn search than the immediate-evaluation variant. For the second-level search it is efficient to use the *immediate-evaluation* variant of the pn-search algorithm for trees, because the second-level pn search uses a fast evaluation function.

If the evaluation by the second-level pn search yields *unknown*, the most-proving node of the first-level search should be expanded, because delayed evaluation is used. However, this node has just been expanded by the second-level pn search. Hence, after completion of the second-level pn search, the children of the root of the second-level search tree (the most-proving node of the first-level search tree) are preserved, but the subtrees of the children are removed. In this way, whenever a most-proving node evaluates to *unknown*, it has already been expanded by the second-level search. If the evaluation by the second-level is *true* or *false* (solving the most-proving node of the first-level search) the second-level search tree is removed completely.

The following important question arises: how many nodes should the second-level pn search use for the evaluation of the most-proving node of the first-level pn search? An attempt to answer this question is made in the next sections.

4.2 The size of the second-level pn search

In this section we investigate how many nodes the second-level pn search should use for the evaluation of the most-proving node of the first-level search. It is not advisable for this number to be large, when the first-level pn search is still small,

because the evaluation of the most-proving nodes then proportionally consumes too much time. Hence, the size of the second-level search tree should be in relation to the size of the first-level search tree.

Allis suggested making this size *equal* to the size of the first-level search tree. From this position he made two statements which are paraphrased below.

1. A search resulting in a first-level tree of size N has searched approximately $\frac{1}{2} \times N^2$ nodes in the second-level search. This can be shown easily by investigating successive steps in the search process. First, the root is evaluated using a second-level search of one node. Then, the root is expanded, and the first new node is evaluated using a second-level search of at least two nodes (depending on the number of children of the root), *etc.* A first-level tree of size N has therefore searched at most $\sum_{i=1}^{N} i = \frac{N \times (N+1)}{2}$ nodes in the second-level search, which, for big N, is approximately equal to $\frac{1}{2} \times N^2$.

2. The memory requirements during the creation of a first-level tree of size N are $2 \times N$ nodes. This is trivial, since the size of the second-level search is set equal to the size of the first-level tree (being N). Therefore, at most $N + N$ nodes are needed to search a first-level tree with N nodes.

A new idea

Allis' suggestion has the disadvantage that relatively easy problems will take much longer to be solved than with standard pn search (see the first statement above). Therefore, we introduce the following idea: start searching with the standard pn-search algorithm; only when it appears that the solution will not be found, start using a second-level search with *growing* size. In this case, solutions of easy problems will still be found fast, and solutions of more difficult problems may also be found because of the increase in directing knowledge since the second-level search tree grows. In conclusion, we suggest that the size of the second-level search tree is some fraction (between 0 and 1) of the size of the first-level search tree. This fraction should preferably start small, and grow larger as the size of the first-level search tree increases.

Let $f(x)$ be a function that determines the fraction, x being the size (i.e., the number of nodes) of the first-level search tree. A standard model for the desired type of growth of the second-level search tree is the logistic-growth model (Berkey, 1988). From this model we adopt the following function

$$f(x) = \frac{1}{1 + e^{(a-x)/b}} \tag{4.1}$$

with two parameters a and b, both strictly positive. The parameter a determines the transition point of the function: as soon as the size of the first-level search tree reaches a nodes, the second-level search uses half the size of the first-level search tree (the larger a, the later this occurs). Parameter b determines the S-shape in the

function (the larger b, the more stretched the S-shape is). We note that the pn^2-search algorithm as suggested by Allis (1994) is a special case of this function: the size of the second-level search tree is equal to the size of the first-level search tree when both parameter a and parameter b become small, because then the fraction function approaches $f(x) = 1$. When a becomes large and b becomes small the fraction function approaches $f(x) = 0$, which means that standard pn search is used[1].

Preliminary experiments revealed that it is advisable to choose the value of parameter a in the order of magnitude of the maximum number of nodes. If parameter a is chosen too small, the transition point moves too far to the left side of the graph (see Figure 4.1), meaning that easy problems will not be found fast any more. If parameter a is chosen too large, the transition point moves too far to the right side of the graph, meaning that too few nodes are used for the second-level pn search, which does not improve the directing knowledge. In this case, the resulting pn^2-search algorithm will have the same drawback as the standard pn-search algorithm, viz. the memory will be filled before a solution is found. Parameter b may have any positive value.

The fraction function exemplified

Figure 4.1 presents four sample functions (with different parameters a and b), illustrating the functions given by Equation 4.1, together with the function $f(x) = 1$. The x-axis shows the number of nodes in the first-level search tree (in thousands). The y-axis shows the corresponding values of the function $f(x)$. Since the pn^2 algorithm will be used when the amount of memory available is low, we assume that no more than 300,000 (300K) nodes fit in memory. Therefore, the range of the x-axis is chosen from 0 to 300K nodes.

The figure shows that when parameter a increases (in this case from 100K to 150K), the transition point moves to the right (compare [4] and [1]). Further, when parameter b increases, the S-shape becomes more stretched (compare [3] and [1]). If b is relatively large, the S-shape may even disappear (compare [2] and [1]).

The theoretical size of the second-level tree

The sizes of the corresponding second-level searches for the five functions of Figure 4.1 are shown in Figure 4.2. The x-axis again shows the number of nodes in the first-level search tree (in thousands). The y-axis shows the size of the corresponding second-level search tree, given by $x \times f(x)$.

From the figure it follows that the size of the second-level search tree grows with increasing size of the first-level search tree. When parameter a increases, the growth of the second-level search tree starts at a later point (compare [4] and [1]). Further, when parameter b increases, the growth of the second-level search tree starts at an earlier point, but the increase becomes slower (compare [3], [1] and [2]).

[1] We note that in our implementation it results in the delayed-evaluation variant of standard pn search.

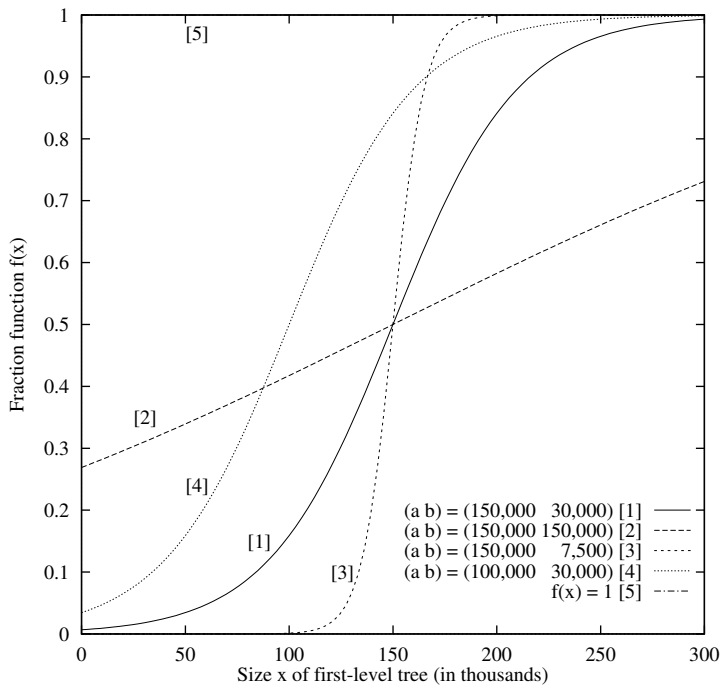

Figure 4.1: The fraction function $f(x)$.

The practical size of the second-level tree

In practice, the sizes of the second-level searches are bounded by the maximum number of nodes that fit into memory (in our case 300K). For instance, if the first-level search tree contains 225K nodes and function $f(x) = 1$ is used, then the second-level search tree should search 225K nodes as well. However, the maximum number of nodes that fit into memory is 300K. Therefore, in this case the second-level search tree has a maximum size of 75K. As soon as this size is reached, the second-level search is terminated, the second-level tree is disposed of, and the first-level search continues. When the first-level tree has reached a size of 300K nodes, no memory is left for the second-level search. In this case, the complete search is terminated and it is indicated that no solution is found.

If these memory bounds are taken into account, the five functions given in Figure 4.2 transform into the functions illustrated in Figure 4.3. The axes are equal to the axes in Figure 4.2

The figure shows that the sizes of the second-level searches are bounded by 300K$-x$.

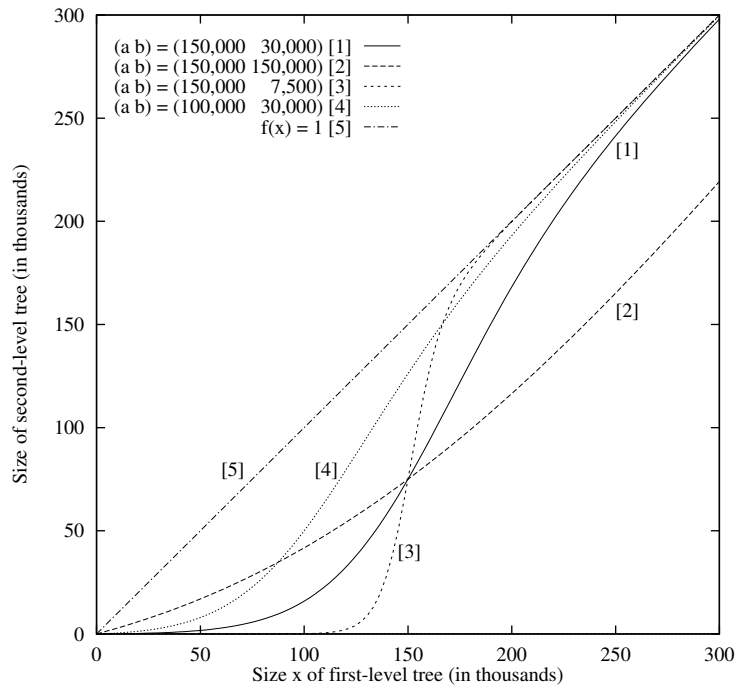

Figure 4.2: The theoretical size of the second-level search.

4.3 Experiments

In this section we describe three series of experiments. They are performed to investigate the behaviour of the pn^2-search algorithm using function $f(x)$ with different parameters a and b. In the first series of experiments we examine the effect of varying parameter b, while keeping parameter a constant. Then, in the second series of experiments we examine the effect of varying parameter a, while keeping parameter b constant. Finally, in the third series of experiments we examine the effect of varying the ratio $\frac{b}{a}$. The idea of examining this ratio stems from the following observations. If a increases, the transition point of the fraction function $f(x)$ shifts to the right. This means that a large first-level tree in memory does not have much information per leaf, because small second-level searches are used for the evaluations. If the first-level tree contains many nodes, the problem to be solved may be a difficult problem, and more directing knowledge may be needed to solve the problem. Therefore, it is then advisable to increase the directing knowledge by also increasing parameter b. Analogously, if a decreases, the transition point of the fraction function $f(x)$ moves to the left in the graph. This means that a small first-level tree in memory already contains much information per node, because relatively large second-level searches are used for the evaluations. In order to limit this overhead it can be wise to reduce

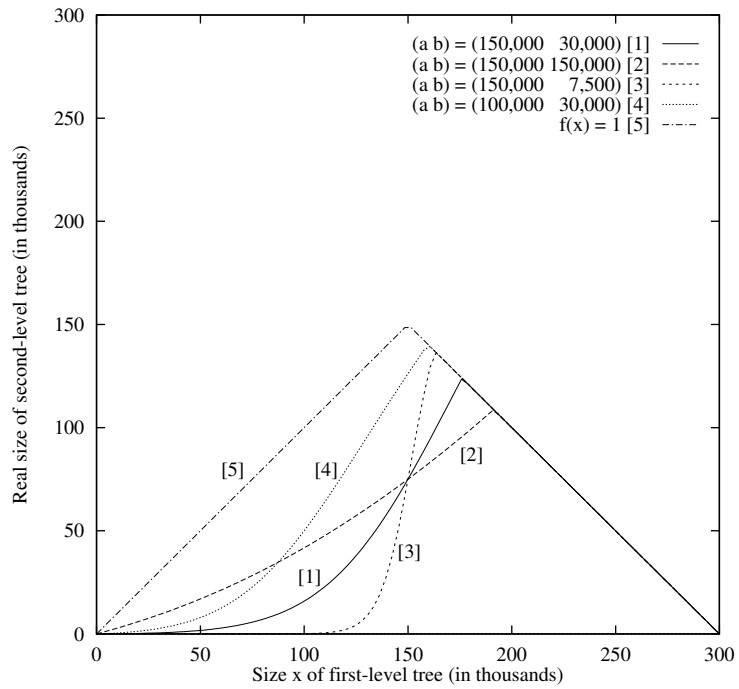

Figure 4.3: The practical size of the second-level search.

the initial size of the second-level search tree, which is taken care of by decreasing parameter b. In both cases the ratio roughly remains equal.

The experiments are performed with second-level searches of size $x \times f(x)$, with $f(x)$ given in Equation 4.1, and with the special case that the second-level search size is x. Further, the maximum number of nodes to be held in memory is set to 300K. The value of parameter a ranges from 75K ($\frac{1}{4}$ of the maximum number of nodes) to 750K ($2\frac{1}{2}$ times the maximum number of nodes). The value of parameter b ranges from 3,750 to 750K. These values were found by trial and error.

The test set for the experiments is a large subset (108 positions) of the set used in the pn-search experiments (see Section 3.4). The positions not tested are K8, K40, K44, K78, K195, K209, K210, K217, and K220[2].

4.4 Results

In this section we discuss the most important results of the experiments mentioned in the previous section. The complete results can be found in Appendix F. We mention

[2] These positions were not solved within 1,000,000 nodes in the previous chapter, and we did not expect that they could be solved within the experimental bounds of this chapter.

that in almost all cases pn^2 search solves all 108 test positions, contrary to pn search for trees (cf. Chapter 3). Therefore, we use as a measure the total amount of nodes searched (i.e., including both first-level and second-level searches) over all 108 test positions.

Results of the first series

Figure 4.4 shows the results of the first series of experiments (varying parameter b, while keeping parameter a constant). Parameter a takes values of 75K, 150K, 300K, 450K, 600K, and 750K, and for each value of a parameter b takes values of 15K, 30K, 60K, 90K, 120K, 150K, 180K, 210K, and 240K. The results of the experiments with function $f(x) = 1$ are shown for comparison. The x-axis shows the value of parameter b (in thousands), and the y-axis shows the total number of nodes searched (in millions).

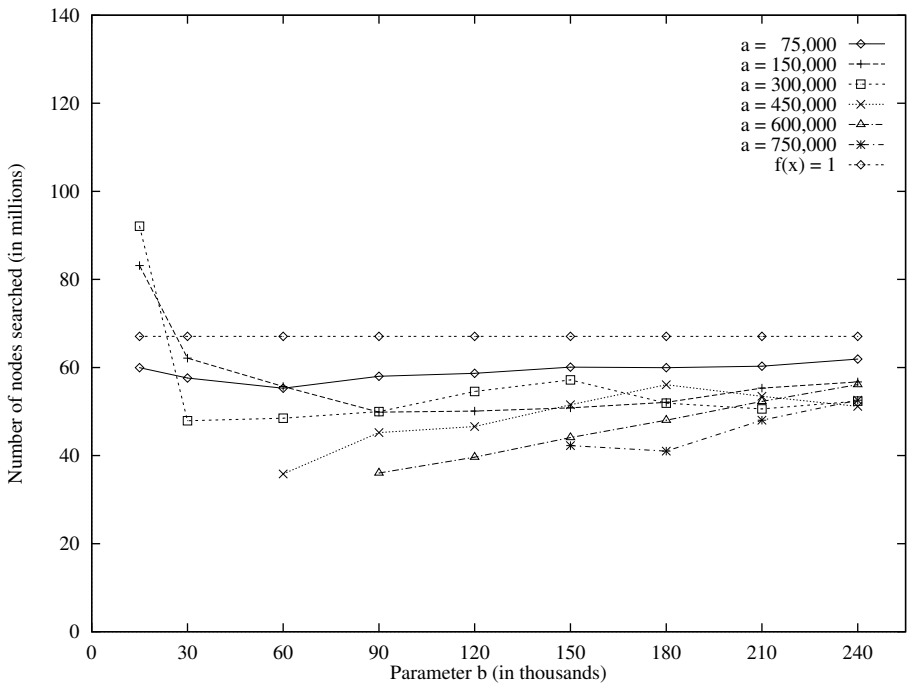

Figure 4.4: The pn^2 results with fixed parameter a.

We note that in the experiments with (a b) equal to (450K 15K), (450K 30K), (600K 15K), (600K 30K), (600K 60K), (750K 15K), (750K 30K), (750K 60K), and (750K 90K) not all 108 test positions are solved. Therefore, these points are not shown in the figure. The number of positions solved in these cases is 88, 104, 87, 91, 106, 87, 87, 99, and 107, respectively. In the positions not solved, the first-level

search tree contained 300K nodes without finding a solution. In these cases the nodes in the first-level search tree do not have sufficient information to direct the search, because the transition point lies too far to the right in the graph (large parameter a).

Results of the second series

The results of the second series of experiments (varying parameter a, while keeping parameter b constant) are shown in Figure 4.5. The x-axis shows the value of parameter a (in thousands). Further explanation for Figure 4.5 is analogous to that for Figure 4.4.

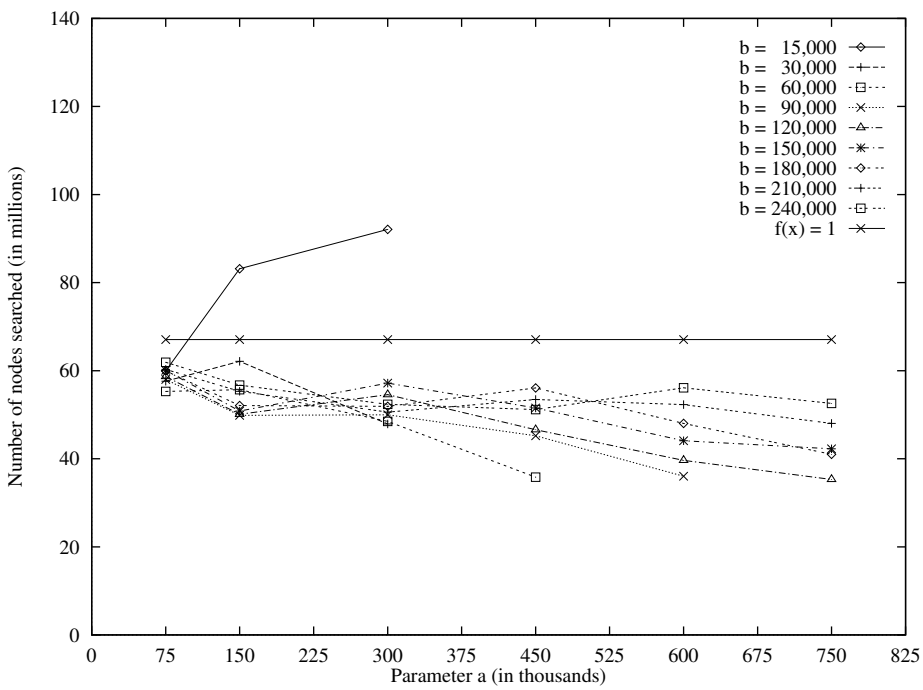

Figure 4.5: The pn^2 results with fixed parameter b.

From these two series of experiments we provisionally conclude that a small parameter b is to be preferred in terms of the number of nodes searched. However, we note that there is a risk of choosing b too small, in which case pn^2 search will not always find a solution. Also, it shows that a large parameter a is to be preferred. Further, the proper use of fraction function $f(x)$ given by the logistic-growth model is significantly better than the function $f(x) = 1$.

Results of the third series

In figure 4.6 the results of the third series of experiments are given (fixed ratio $\frac{b}{a}$). The ratio takes the values 0.05, 0.1, 0.2, 0.4, 0.6, 0.8 and 1.0. Again, the results of the experiments with function $f(x) = 1$ are shown for comparison. We note that in the experiments with (a b) equal to (450K 22.5K), (600K 30K), (600K 60K), (750K 37.5K), and (750K 75K) not all test positions are solved. Therefore, these points are not shown in the figure. The number of positions solved in these cases is 98, 91, 106, 87, and 104, respectively.

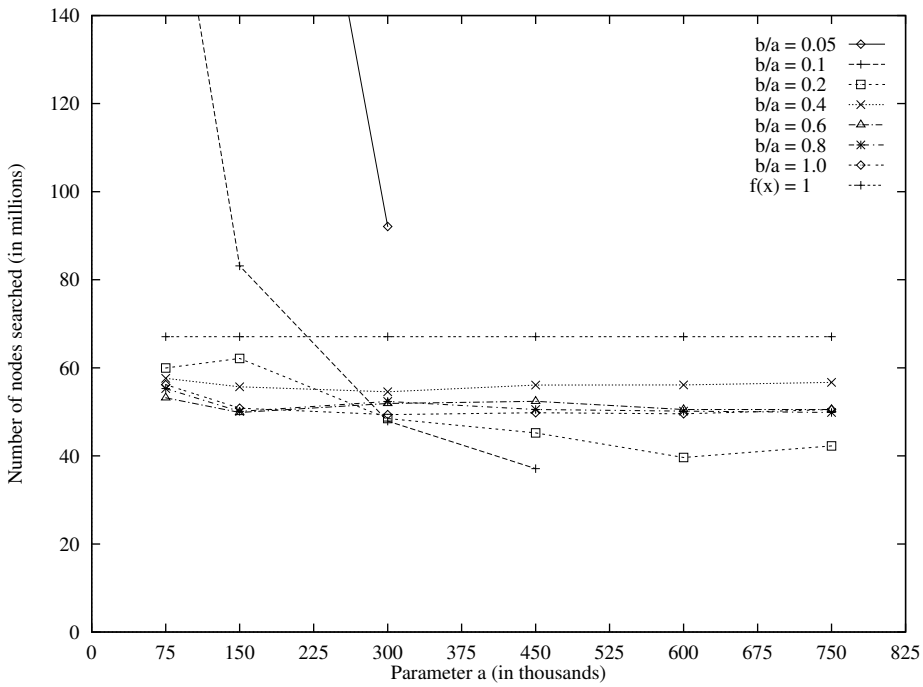

Figure 4.6: The pn^2 results with fixed ratio $\frac{b}{a}$.

From this graph we conclude that, with a ratio of more than 0.1, the results seem to be fairly independent of the choice of the parameters.

4.5 Chapter conclusions

For the pn^2 algorithm we conclude that the use of the function $f(x) = 1$ works well, since it solves all test positions unlike standard pn search. However, the use of fraction function $f(x)$ as given by Equation 4.1, gives significantly better results.

If the goal of the search is to find a solution as quickly as possible, it is recommended to take the fraction function with large parameter a and small parameter b. The disadvantage is that sometimes solvable positions will not be solved because the standard pn search takes too long, filling the memory with nodes not containing sufficient directing knowledge.

If the goal of the search is to solve any solvable position, it is wiser to choose parameters a and b, such that $\frac{b}{a}$ is sufficiently large, i.e., at least more than 0.1. The best performance on the test set is obtained by choosing (a b) equal to (600K 80K) (cf. Appendix F). All test positions are then solved within about 35 million nodes.

Of the 108 test positions, 92 were solved by both the immediate-evaluation variant of standard pn search and the best version of pn^2 search (a = 600K and b = 80K). For these 92 positions the number of nodes searched by pn^2 search (5,856,337) is about twice the number of nodes searched by standard pn search (2,974,602). In our view this is an affordable price for the advantage of the larger probability of finding mates (in this case amounting to an additional 18% solved positions). Further, for these 92 positions pn^2 search used at maximum about 240K (first-level and second-level) nodes in memory.

The results of the experiments in this chapter show that the pn^2-search algorithm is an adequate method for reducing the need for memory in the standard pn-search algorithm. This is accomplished by gaining more knowledge per node through increasing the search: leaves are evaluated using a second-level pn search. The use of the growth-function $f(x)$ proposed here gives significantly better results than the naïve implementation of pn^2 search (effectively using $f(x) = 1$).