Towards Distributed Algorithm Portfolios

Matteo Gagliolo[1,2] and Jürgen Schmidhuber[1,2,3]

[1] IDSIA, Galleria 2, 6928 Manno (Lugano), Switzerland
[2] University of Lugano, Faculty of Informatics, Via Buffi 13, 6904 Lugano, Switzerland
[3] TU Munich, Boltzmannstr. 3, 85748 Garching, München, Germany
{matteo,juergen}@idsia.ch

Summary. In recent work we have developed an online algorithm selection technique, in which a model of algorithm performance is learned incrementally *while* being used. The resulting *exploration-exploitation* trade-off is solved as a bandit problem. The candidate solvers are run in parallel on a single machine, as an *algorithm portfolio*, and computation time is shared among them according to their expected performances. In this paper, we extend our technique to the more interesting and practical case of multiple CPUs.

1 Introduction

Existing parallel computing systems (e. g., Condor, Globus, JOpera), are aimed at improving speed and reliability of computation, but require the user to specify which computations should be carried out. In a more practical situation, a set of alternative algorithms, of unknown performance, is available for a given problem class, and one would like to automate the process of selecting which algorithm to use, independently for each problem instance.Such automatic *algorithm selection* is now a thirty years old field of artificial intelligence [19], even though most work in this area has been done in the last two decades. In the more general *algorithm portfolio* paradigm [16],the available algorithms are executed in parallel, and the aim of selection is to allocate computational resources among them. The problem of algorithm (portfolio) selection consists in deciding which experiments should be carried out, given a fixed amount of computational resources. In parallel computing, the set of experiments is fixed by the user, while the available computational resources may vary over time, due to failures, load fluctuations, node addition, etc. In both fields, the aim is obviously to minimize computation time, and to decrease the amount of human work and expertise required. With this paper, we intend to move a step towards the integration of these two orthogonal approaches, devising the blueprint of a more "intelligent" cluster front-end. In the following section we will briefly review related work. Section 3 describes our algorithm selection framework GAMBLETA. Section 4 discusses its extension to the case of multiple CPUs.After reporting experimental results (Section 5), we conclude the paper in Section 6.

2 Related work

In general terms, algorithm selection can be defined as the process of allocating computational resources to a set of alternative algorithms, in order to improve some measure of performance on a set of problem instances. For *decision* or *search* problems, where a binary criterion for recognizing a solution is available (e. g., SAT [10]), the only meaningful measure of performance is runtime, and selection is aimed at minimizing it. The selection among different algorithms can be performed once for an entire set of problem instances (*per set* selection); or repeated for each instance (*per instance* selection). As in most practical cases there is no single "best" algorithm, per instance selection can only improve over per set selection. A further orthogonal distinction can be made among *static* algorithm selection, in which any decision on the allocation of resources precedes algorithm execution; and *dynamic* algorithm selection, in which the allocation can be adapted during algorithm execution. Selection based on a model of algorithm performance can be further distinguished as *offline*, if performance data is gathered during a preliminary training phase, after which the model is kept fixed; or *online*, if the model is updated at every instance solution.

A seminal paper in this field is [19], in which offline, per instance selection is first advocated. More recently, similar concepts have been proposed by the *Meta-Learning* community [7, 20]. *Parameter tuning* can also be modeled as an algorithm selection problem: in this case the algorithm set is composed of multiple copies of the same algorithm, differing only in the parameter values.

The foundation papers about algorithm portfolios [16, 14, 12] describe how to evaluate the runtime distribution of a portfolio, based on the runtime distributions of the algorithms.The RTD is used to evaluate mean and variance, and find the (per set optimal) *efficient frontier* of the portfolio, i.e., that subset of all possible allocations in which no element is dominated in both mean and variance. Another approach based on runtime distributions can be found in [4, 5], for parallel independent processes and shared resources respectively. The expected value of a cost function, accounting for both wall-clock time and resources usage, is minimized. In all these works the runtime distributions are assumed to be known *a priori*.

Further references on algorithm selection can be found in [8, 9]. Literature on parallel computing, grid computing, distributed computing [6, 1, 18] is focused on allocation of dynamically changing computational resources, in a transparent and fault tolerant manner. We are not aware of works in this field in which algorithm selection is considered. Performance modeling can be used to guide scheduling decisions, as in [1], where a simple estimate of expected runtime is performed, in order to be able to meet user-imposed deadlines.

3 Allocating a single CPU

Online model-based algorithm selection consists in updating a model of performance *while* using it to guide selection. It is intuitive that this setting poses what is called an *exploration-exploitation* trade-off: on the one hand, collecting runtime data for

improving the model can lead to better resource allocation in the future. On the other hand, running more experiments to improve a model which already allows to perform good selections can be a waste of machine time. A well known theoretical framework for dealing with such a dilemma is offered by the the *multi-armed bandit* problem [2], consisting of a sequence of trials against a K-armed slot machine. At each trial, the gambler chooses one of the available arms, whose losses are randomly generated from different unknown distributions, and incurs in the corresponding loss. The aim of the game is to minimize the *regret*, defined as the difference between the cumulative loss of the gambler, and the one of the best arm. A bandit problem solver (BPS) can be described as a mapping from the history of the observed losses l_k for each arm k, to a probability distribution $\mathbf{p} = (p_1, ..., p_K)$, from which the choice for the successive trial will be picked. Given the notion of regret, a straightforward application of a BPS to algorithm selection, in which "pick arm k" means "run algorithm a_k on next problem instance", would only allow to identify the per-set optimal algorithm [8]. Suppose instead we do have a model based method for *per-instance* algorithm selection, and we only want to know when the model is reliable enough. We can then play the game at an upper level, choosing, for each subsequent problem instance, between the model based selector and a simpler and more exploratory allocation strategy, such as a parallel portfolio of all available algorithms. Intuitively, the BPS will initially penalize the model-based allocator, but only until the model is good enough to outperform the exploratory allocator. Alternative allocation techniques can be easily added as additional "arms" of the bandit.

More precisely, consider a sequence $\mathcal{B} = \{b_1, \ldots, b_M\}$ of M instances of a decision problem, for which we want to minimize solution time; a set of K algorithms $\mathcal{A} = \{a_1, \ldots, a_K\}$; and a set of N *time allocators* (TA_j) [8]. Each TA_j can be an arbitrary function, mapping the current history of collected performance data for each a_k, to a share $\mathbf{s}^{(j)} \in [0, 1]^K$, with $\sum_{k=1}^{K} s_k = 1$. A TA is used to solve a given problem instance executing all algorithms in $\mathcal{A}$ in parallel, on a single machine, whose computational resources are allocated to each a_k proportionally to the corresponding s_k, such that for any portion of time spent t, $s_k t$ is used by a_k, as in a *static* algorithm portfolio [16]. The runtime before a solution is found is then $\min_k\{t_k/s_k\}$, t_k being the runtime of algorithm a_k when executed alone. A trivial example of an exploratory TA is the *uniform* time allocator, assigning a constant $\mathbf{s} = (1/K, ..., 1/K)$. Single algorithm selection can be represented in this framework by setting a single s_k to 1. Dynamic allocators will produce a time-varying share $\mathbf{s}(t)$. The TAs proposed in [8] are based on non-parametric models of the runtime distribution of the algorithms, which are used to optimize the share $\mathbf{s}$ according to different criteria (see next section).

At this higher level, one can use a BPS to select among different time allocators, $\text{TA}_j, \text{TA}_2 \ldots$, working on a same algorithm set $\mathcal{A}$. In this case, "pick arm j" means "use time allocator TA_j on $\mathcal{A}$ to solve next problem instance". In the long term, the BPS would allow to select, on a *per set* basis, the TA_j that is best at allocating time to algorithms in $\mathcal{A}$ on a *per instance* basis. The resulting "Gambling" Time Allocator (GAMBLETA) is described in Alg. 1.

Algorithm 1 GAMBLETA($\mathcal{A}$,$\mathcal{T}$,BPS) Gambling Time Allocator.

Algorithm set $\mathcal{A}$ with K algorithms;
A set $\mathcal{T}$ of N time allocators TA$_j$;
A bandit problem solver BPS
M problem instances.
initialize BPS(N, M)
for each problem $b_i, i = 1, \ldots, M$ **do**
  pick time allocator $I(i) = j$ with probability $p_j(i)$ from BPS.
  solve problem b_i using TA$_I$ on $\mathcal{A}$
  incur loss $l_{I(i)} = \min_k\{t_k(i)/s_k^{(I)}(i)\}$
  update BPS
end for

One motivation for using a BPS is the guaranteed bound on regret. In [9], basing on [3], we introduced EXP3LIGHT-A, a BPS which guarantees a bound on regret when the maximum loss is unknown *a priori*. Note that any bound on the regret of the chosen BPS will determine a bound on the regret of GAMBLETA with respect to the best time allocator. Nothing can be said about the performance w.r.t. the best algorithm. In a worst-case setting, if none of the time allocator is effective, a bound can still be obtained by including the uniform share in the set of TAs. In practice, though, per-instance selection can be much more efficient than uniform allocation, and the literature is full of examples of time allocators converging to a good performance.

4 Allocating multiple CPUs

For simplicity, we assume a traditional cluster setup, with a front end controlling the allocation of jobs on different nodes. Consider again our K algorithms $\mathcal{A} = \{a_1, a_2, ..., a_K\}$. This time we need to allocate time on J CPUs, so the share will be represented as a $K \times J$ matrix, with columns summing to 1, $\mathbf{S} = \{s_{kj}\}, s_{kj} \in [0, 1], \sum_{k=1}^{K} s_{kj} = 1 \forall j \in \{0, 1, \ldots, J\}$. For a given share $\mathbf{S}$, the *survival function*[4] for the distributed portfolio can be evaluated as (compare with (12) from [8]):

$$S_{\mathcal{A},\mathbf{S}}(t) = \prod_{j=1}^{J} \prod_{k=1}^{K} S_k(s_{kj}t), \quad (1)$$

We will first see how to apply the model-based time allocators, introduced in [8, Sec. 4] for a single CPU, to a case in which $J > 1$ CPUs are used:

1. **Expected time.** The expected runtime value is minimized w.r.t. $\mathbf{S}$:

$$\mathbf{S}^* = \arg\min_{\mathbf{S}} \int_0^{+\infty} S_{\mathcal{A},\mathbf{S}}(t)dt. \quad (2)$$

[4] $S(t) = 1 - F(t)$, F being the cumulative distribution function of the runtime distribution.

2. **Contract.** This TA picks the $\mathbf{S}$ that maximizes the probability of solution within a *contract* time t_u , or, equivalently, minimizes the survival function at t_u:

$$\mathbf{S}^*(t_u) = \arg\min_{\mathbf{S}} S_{\mathcal{A},\mathbf{S}}(t_u). \tag{3}$$

3. **Quantile.** This TA minimizes a *quantile* α of (1):

$$\mathbf{S}^*(\alpha) = \arg\min_{\mathbf{S}} F^{-1}_{\mathcal{A},\mathbf{S}}(\alpha). \tag{4}$$

While in [8] the share s was found optimizing functions in a $(K-1)$ dimensional space, here the size of the search space grows also with the number of CPUs, as $J(K-1)$. Fortunately, and rather unexpectedly, we could prove that the optimal share for the contract and quantile allocators is *homogeneous*, i. e., the same on each CPU, such that the corresponding matrix has all its columns equal (see Appendix). This means that it can be found with a search in a $(K-1)$ dimensional space, regardless of the number of CPUs available.

The above criteria assume that one wants to use all available CPUs. This may or not be convenient, depending on the shape of the runtime distributions at play. One simple possibility for selecting the number of CPUs is to optimize the allocation for $1, 2, \ldots$ up to J CPUs, and then use the number j that gives the best result, considering that if a problem is solved in a time t using j CPUs, the total CPU time used will be jt.

A *dynamic* version of the above TAs can be implemented, periodically updating the model, for example conditioning on the time already spent as in [8], and re-evaluating the optimal share. Regarding the number of CPUs allocated, this should only be decreased, as increasing it would require to start the algorithms from scratch, with the unconditioned model: in this situation the optimal share would not be homogeneous, and we would have again a larger search space.

In the experiments described in the next section, we used a set of quantile allocators, with different parameter values for α, along with the uniform allocator, and considered the following heuristic: for each problem instance, the front end picks a time allocator, allowing it the use of all currently available CPUs J. The uniform allocator will take them all, and run all algorithms in parallel on each, obviously with a different random seed. The quantile allocators will instead evaluate the optimal share for $1, 2, \ldots, J$ CPUs, and pick a number j of them such that jt_α is minimized, $t_\alpha = F^{-1}(\alpha)$ being the quantile. The front-end will continue assigning the remaining $J - j$ CPUs for the next problem instance, until all CPUs are occupied. When a quantile TA dynamically updates its share, it re-evaluates it for $1, 2, \ldots, j$ CPUs, releasing some of the CPUs if necessary. Released CPUs are then reallocated by the front-end to solve the following problem instance. At the upper level, the BPS (EXP3LIGHT-A from [9]) is used as in GAMBLETA, using the *total* CPU time as a loss (i. e., jt if j CPUs are used for a wall-clock time t), in order to favor time allocators that do not use more CPUs than necessary.

Unfortunately, the bound on regret of the BPS does not hold in the case of multiple CPUs, as the proof assume a *sequential* game, in which the probability distribution over the arms is updated after each arm pull, based on the observed loss.

In this multiple CPU version of GAMBLETA, instead, the choice of time allocators continues until there are CPUs available, and the feedback on the loss is received asynchronously, only when the corresponding problem instance is solved.

5 Experiments

The main objective of these preliminary experiments is to analyze the speedup (the ratio between runtime with 1 and $J > 1$ CPUs) and efficiency (the ratio between speedup and number of CPUs) of the proposed allocation method. Note that the notion of efficiency assumes a different connotation in the context of algorithm portfolios: traditionally one does not expect to achieve an efficiency larger than 1, as it is assumed that all computations performed on a single CPU have to be carried out on J CPUs as well. This is not the case for algorithm portfolios, as in this case we can stop the computation as soon as the fastest algorithm solves the problem, so we will see efficiencies greater than 1.

In the first experiment we apply GAMBLETA to a solve a set of satisfiable and unsatisfiable CNF3SAT problems (benchmarks `uf-*`, `uu-*` from [15], 1899 instances in total), using a small algorithm set composed of a local search and a complete SAT solver (respectively, G2-WSAT [17] and Satz-Rand [13]). Local search algorithms are more efficient on satisfiable instances, but cannot prove unsatisfiability, so are doomed to run forever on unsatisfiable instances; while complete solvers are guaranteed to terminate their execution on all instances, as they can also prove unsatisfiability. The set of time allocators includes the uniform one, and nine quantile allocators, with α ranging from 0.1 to 0.9, sharing the same conditional non-parametric model from [21]. As the clauses-to-variable ratio is fixed in this benchmark, only the number of variables, ranging from 20 to 250, was used to condition the model.

In a second set experiment we use Satz-Rand alone on 9 sets of structured graph-coloring (GC) problems [11], also from [15], each composed of 100 instances encoded in CNF 3 SAT format. This algorithm/benchmark combination is particularly interesting as the *heavy-tailed* behavior [13] of Satz-Rand differs for the various sets of instances[5] [11]. In this case, the time allocators decide only how many parallel copies of Satz-Rand to run for each problem: as the share is 1 on each CPU, we allowed the TAs to dynamically shrink and also grow the number of CPUs used.

Both experiments were simulated in MATLAB, on a single machine, based on previously collected runtime data[6], for different numbers of CPUs $(1, 5, 10, 15, 20)$.

---

[5] A *heavy-tailed* runtime distribution $F(t)$ is characterized by a Pareto tail, i.e., $F(t) \to_{t\to\infty} 1 - Ct^{-\alpha}$. In practice, this means that most runs are relatively short, but the remaining few can take a very long time, with differences among runs of several orders of magnitude. In this situation, both restart strategies and parallel execution proved to be an effective way of reducing runtime [12].

[6] As we needed a common measure of time, and the CPU runtime measures are quite inaccurate, we modified the original code of the two algorithms adding a counter, that is incremented at every loop in the code. All runtimes reported for this benchmark are expressed in these loop cycles: on a 2.4 GHz machine, 10^9 cycles take about 1 minute.

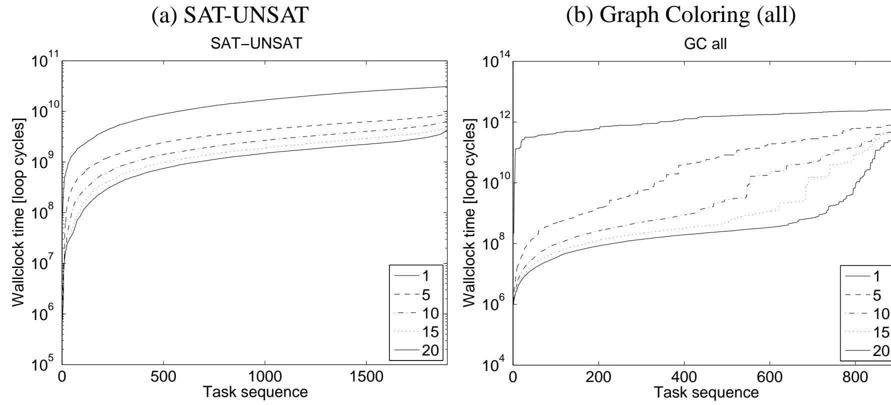

Fig. 1. (a): Wall-clock time for the SAT-UNSAT benchmark, for different numbers of CPUs ($10^9 \approx 1$ min.). (b): Wall-clock time for the Graph Coloring benchmark (all problems), for different numbers of CPUs ($10^9 \approx 1$ min.).

	Speedup				Efficiency			
#CPUs	5	10	15	20	5	10	15	20
GC 0	4.34	7.19	9.23	11.19	0.87	0.72	0.62	0.56
GC 1	1.00	279.26	83.55	85.87	0.20	27.93	5.57	4.29
GC 2	2.65	18.44	35.98	97.74	0.53	1.84	2.40	4.89
GC 3	2.59	6.55	9.80	21.28	0.52	0.66	0.65	1.06
GC 4	3.97	6.18	6.94	8.81	0.79	0.62	0.46	0.44
GC 5	3.36	3.47	7.69	7.83	0.67	0.35	0.51	0.39
GC 6	3.55	5.13	5.79	8.54	0.71	0.51	0.39	0.43
GC 7	5.93	8.27	12.65	11.95	1.19	0.83	0.84	0.60
GC 8	5.06	9.02	11.62	12.01	1.01	0.90	0.77	0.60
GC all	3.06	4.46	5.50	8.22	0.61	0.45	0.37	0.41
SAT-UNSAT	3.46	4.81	6.02	7.08	0.69	0.48	0.40	0.35

Table 1. Speedup (on the left) and efficiency (on the right) for the SAT-UNSAT and the different subgroups of Graph Coloring (GC) benchmarks. Note the dramatic speed-up obtained on GC sets 1 and 2, the effect of the heavy-tailed RTD of Satz-Rand on these problem sets.

Results reported are upper confidence bounds obtained from 20 runs, each time using fresh random seeds, and a different random reordering of the problem instances.

6 Conclusions

We presented a framework for distributed time allocation, aimed at minimizing solution time of decision problems. Preliminary results are encouraging, even though the efficiency sometimes decreases with the number of CPUs.

The difference with the "embarrassingly parallel" computation paradigm, in which all processes can be executed independently, is that in our case there is an indirect interaction among the algorithms, as resource allocation among elements of the algorithm set is periodically updated, based on runtime information (*dynamic* selection); and as soon as one algorithm solves a given problem, other algorithms working on the same problem are stopped. Note that, as our algorithm selection scheme has a very low computational overhead [8], it can be easily extended to a fully distributed situation, in which there is no front end, and all nodes are equal. In such case, runtime data can be broadcast, in order to allow each node to update a local copy of the RTD model; as the time allocation algorithm is pseudo-random, it can be reproduced deterministically, so each node can independently evaluate the same allocation, and execute the job(s) assigned to itself. Existing distributed computing techniques can be used at a lower level, to deal with message losses and node failures.

Acknowledgments. This work was supported by the Hasler foundation with grant n. 2244.

References

1. David Abramson, Jonathan Giddy, and Lew Kotler. High performance parametric modeling with Nimrod/G: Killer application for the global grid? In *Proceedings of the 14th International Parallel & Distributed Processing Symposium (IPDPS'00), Cancun, Mexico, May 1-5, 2000*, pages 520–528, 2000.
2. Peter Auer, Nicolò Cesa-Bianchi, Yoav Freund, and Robert E. Schapire. The nonstochastic multiarmed bandit problem. *SIAM J. Comput.*, 32(1):48–77, 2002.
3. Nicolò Cesa-Bianchi, Yishay Mansour, and Gilles Stoltz. Improved second-order bounds for prediction with expert advice. In Peter Auer et al., editors, *COLT*, volume 3559 of *Lecture Notes in Computer Science*, pages 217–232. Springer, 2005.
4. Lev Finkelstein, Shaul Markovitch, and Ehud Rivlin. Optimal schedules for parallelizing anytime algorithms: the case of independent processes. In *Eighteenth national conference on Artificial intelligence*, pages 719–724. AAAI Press, 2002.
5. Lev Finkelstein, Shaul Markovitch, and Ehud Rivlin. Optimal schedules for parallelizing anytime algorithms: The case of shared resources. *Journal of Artificial Intelligence Research*, 19:73–138, 2003.
6. I. Foster and C. Kesselman. Globus: A metacomputing infrastructure toolkit. *The International Journal of Supercomputer Applications and High Performance Computing*, 11(2):115–128, Summer 1997.
7. Johannes Fürnkranz. On-line bibliography on meta-learning, 2001. EU ESPRIT METAL Project (26.357): A Meta-Learning Assistant for Providing User Support in Machine Learning Mining.
8. Matteo Gagliolo and Jürgen Schmidhuber. Learning dynamic algorithm portfolios. *Annals of Mathematics and Artificial Intelligence*, 47(3–4):295–328, August 2006. AI&MATH 2006 Special Issue.
9. Matteo Gagliolo and Jürgen Schmidhuber. Algorithm selection as a bandit problem with unbounded losses. Technical Report IDSIA-07-08, IDSIA, 2008.
10. I. Gent and T. Walsh. The search for satisfaction. Technical report, Dept. of Computer Science, University of Strathclyde, 1999.

11. Ian Gent, Holger H. Hoos, Patrick Prosser, and Toby Walsh. Morphing: Combining structure and randomness. In *Proc. of AAAI-99*, pages 654–660, 1999.
12. Carla P. Gomes and Bart Selman. Algorithm portfolios. *Artificial Intelligence*, 126(1–2):43–62, 2001.
13. Carla P. Gomes, Bart Selman, Nuno Crato, and Henry Kautz. Heavy-tailed phenomena in satisfiability and constraint satisfaction problems. *J. Autom. Reason.*, 24(1-2):67–100, 2000.
14. Carla P. Gomes, Bart Selman, and Henry Kautz. Boosting combinatorial search through randomization. In *AAAI '98/IAAI '98*, pages 431–437, USA, 1998. AAAI Press.
15. H. H. Hoos and T. Stützle. SATLIB: An Online Resource for Research on SAT. In I.P.Gent et al., editors, *SAT 2000*, pages 283–292, 2000. `http://www.satlib.org`.
16. B. A. Huberman, R. M. Lukose, and T. Hogg. An economic approach to hard computational problems. *Science*, 275:51–54, 1997.
17. Chu Min Li and Wenqi Huang. Diversification and determinism in local search for satisfiability. In *SAT2005*, pages 158–172. Springer, 2005.
18. Cesare Pautasso, Win Bausch, and Gustavo Alonso. Autonomic computing for virtual laboratories, 2006.
19. J. R. Rice. The algorithm selection problem. In Morris Rubinoff and Marshall C. Yovits, editors, *Advances in computers*, volume 15, pages 65–118. Academic Press, New York, 1976.
20. Ricardo Vilalta and Youssef Drissi. A perspective view and survey of meta-learning. *Artif. Intell. Rev.*, 18(2):77–95, 2002.
21. Laura Wichert and Ralf A. Wilke. Application of a simple nonparametric conditional quantile function estimator in unemployment duration analysis. Technical Report ZEW Discussion Paper No. 05-67, Centre for European Economic Research, 2005.

Appendix

Definition 1 (Homogeneous share). *Be* $\mathbf{s}_j$ *the share for CPU j (i.e., the j-th column of* $\mathbf{S}$*). A share* $\mathbf{S}$ *is* homogeneous *iff* $s_{kj} = s_k \forall (k, j)$*, i.e.,* $\mathbf{s}_j = \mathbf{s} \forall j$.

It is easy to prove by examples that expected-value-optimal shares can be inhomogeneous. For contract and quantile optimal shares, the following theorems hold.

Theorem 1. $\forall$ *contract-optimal share* $\mathbf{S}^*(t_u)$ *there is an homogeneous equivalent* $\mathbf{S}^*_h(t_u)$ *such that* $S_{\mathcal{A},\mathbf{S}^*}(t_u) = S_{\mathcal{A},\mathbf{S}^*_h}(t_u)$.

Proof.

$$\mathbf{S}^* = \arg\min_{\mathbf{S}} S_{\mathcal{A},\mathbf{S}}(t_u) = \arg\min_{\{\mathbf{s}_j\}} \prod_{j=1}^{J} S_{\mathcal{A},\mathbf{s}_j}(t_u) \tag{5}$$

Consider a non homogeneous contract-optimal share $\mathbf{S}^*(t_u)$, with $\mathbf{s}^*_j \neq \mathbf{s}^*_i$ for a pair of columns i, j. If $S_{\mathcal{A},\mathbf{s}_i}(t_u) > S_{\mathcal{A},\mathbf{s}_j}(t_u)$, then replacing $\mathbf{s}_i$ with $\mathbf{s}_j$ produces a better share, violating the hypothesis of optimality of $\mathbf{S}^*$. As this must hold for any i, j, then $S_{\mathcal{A},\mathbf{s}_i}(t_u)$ must be the same for all i. Setting all $\mathbf{s}_j$ to a same $\mathbf{s}_i$ will then produce a homogeneous optimal share $\mathbf{S}^*_h$.

Theorem 2. $\forall$ *quantile-optimal share* $\mathbf{S}^*(\alpha)$ *there is an homogeneous equivalent* $\mathbf{S}^*_h(\alpha)$ *such that* $F^{-1}_{\mathcal{A},\mathbf{S}^*}(\alpha) = F^{-1}_{\mathcal{A},\mathbf{S}^*_h}(\alpha)$.

Proof. From its definition, a quantile t_α is $t_\alpha = min\{t|F(t) = \alpha\} = min\{t|S(t) = (1-\alpha)\}$; this, together with the monotonicity of the survival function, and of the logarithm function, implies:

$$\ln(1-\alpha) = \ln S(t_\alpha) < \ln S(t) \;\; \forall \;\; t < t_\alpha. \tag{6}$$

Consider a non homogeneous optimal share $\mathbf{S}^*$, with $\mathbf{s}^*_j \neq \mathbf{s}^*_i$ for a pair $(i, j), i \neq j$. We can write:

$$\ln(1-\alpha) = \ln S_{\mathcal{A},\mathbf{S}}(t_\alpha) = \sum_{j=1}^{J} \ln S_{\mathcal{A},\mathbf{s}_j}(t_\alpha) \tag{7}$$

Pick now an arbitrary column $\mathbf{s}^*_i$ of $\mathbf{S}^*$, and set all other columns $\mathbf{s}^*_j$ to $\mathbf{s}^*_i$, obtaining a homogeneous share with quantile t_i

$$\ln(1-\alpha) = J \ln S_{\mathcal{A},\mathbf{s}_i}(t_i) \tag{8}$$

While $t_i < t_\alpha$ violates the hypothesis of optimality of $\mathbf{S}^*(\alpha)$, $t_\alpha < t_i$ would imply for (6) (with t_i in place of t_α):

$$\ln S_{\mathcal{A},\mathbf{s}_i}(t_i) = \frac{\ln(1-\alpha)}{J} < \ln S_{\mathcal{A},\mathbf{s}_i}(t_\alpha). \tag{9}$$

Summing over i gives a contradiction ($\ln(1-\alpha) < \ln(1-\alpha)$), so the only possibility left is $t_i = t_\alpha$. As this must hold for any i, then t_i must be constant $\forall i$. Setting all $\mathbf{s}_j$ to a same $\mathbf{s}_i$ will then produce a homogeneous optimal share $\mathbf{S}^*_h$. Note that, different from the contract, in this case $\mathbf{S}^*_h$ depends on J.