Markovian PERT networks: A new CTMC and benchmark results

Stefan Creemers

(September 6, 2017)

IÉSEG SCHOOL OF MANAGEMENT

KATHOLIEKE UNIVERSITEIT LEUVEN

Kulkarni & Adlakha (1986)

- Markov and Markov-Regenerative PERT Networks, *Operations Research*, 1986
- 208 citations
- First to study Markovian PERT networks
- Use of a CTMC to model a network
- The states of the CTMC are defined by three sets: idle, ongoing, & finished activities

$\Rightarrow$There are up to 3^n states!

$\Rightarrow$Need for a strict partitioning of the statespace!

Creemers, Leus, & Lambrecht (2010)

- Scheduling Markovian PERT networks to maximize the net present value, *Operations Research Letters*, 2010
- Studies the **SNPV**
- First to suggest a strict partitioning of the statespace
- Use of UDCs to only store feasible states
- UDCs have later been adopted by:
  - Wei et al. (2013) *Expert Systems with Applications*
  - Coolen et al. (2015) *Journal of Scheduling*
  - Gutin et al. (2015) *Management Science*
  - Rostami et al. (2017) *Journal of Scheduling*
  - **Creemers (2015) *Journal of Scheduling***

Creemers (2015)

- Minimizing the expected makespan of a project with stochastic activity durations under resource constraints, *Journal of Scheduling*, 2015
- Studies the **SRCPSP**
- Uses CTMC of Kulkarni & Adlakha and the UDCs of Creemers et al. (2010)
- Significantly improves procedure of Creemers et al. (2010)
- To compare with Creemers et al. (2010), we adapt Creemers (2015) to also solve the SNPV

2010 VS 2015
Number of instances solved

2010			
Instances solved			
	OS = 0.8	OS = 0.6	OS = 0.4
n = 10	30	30	30
n = 20	30	30	30
n = 30	30	30	30
n = 40	30	30	29
n = 50	30	30	4
n = 60	30	30	0
n = 70	30	22	0

2015			
Instances solved			
	OS = 0.8	OS = 0.6	OS = 0.4
n = 10	30	30	30
n = 20	30	30	30
n = 30	30	30	30
n = 40	30	30	29
n = 50	30	30	4
n = 60	30	30	0
n = 70	30	22	0

We use the dataset of Creemers et al. (2010) to compare the performance of the 2010 & 2015 procedures.

2010 VS 2015
Average CPU time (sec)

2010			
Average CPU time (s)			
	OS = 0.8	OS = 0.6	OS = 0.4
n = 10	0	0	0
n = 20	0	0	0
n = 30	0	0	27
n = 40	0	7	2338
n = 50	0	100	52268
n = 60	1	2210	NA
n = 70	3	17496	NA

2015			
Average CPU time (s)			
	OS = 0.8	OS = 0.6	OS = 0.4
n = 10	0	0	0
n = 20	0	0	0
n = 30	0	0	2
n = 40	0	1	92
n = 50	0	6	1048
n = 60	0	89	NA
n = 70	0	505	NA

On average, we improve computation times by a factor of 43!

2010 VS 2015

Average number of states (per 1000)

2010			
Average state-space size (per 1000)			
	OS = 0.8	OS = 0.6	OS = 0.4
n = 10	0	0	1
n = 20	0	4	55
n = 30	2	49	1560
n = 40	8	534	47073
n = 50	27	4346	526020
n = 60	92	42279	NA
n = 70	287	216028	NA

2015			
Average state-space size (per 1000)			
	OS = 0.8	OS = 0.6	OS = 0.4
n = 10	0	0	1
n = 20	0	4	55
n = 30	2	49	1560
n = 40	8	534	47073
n = 50	27	4346	526020
n = 60	92	42279	NA
n = 70	287	216028	NA

2010 VS 2015
Bottleneck

2015			
Average CPU time (s)			
	OS = 0.8	OS = 0.6	OS = 0.4
n = 10	0	0	0
n = 20	0	0	0
n = 30	0	0	2
n = 40	0	1	92
n = 50	0	6	1048
n = 60	0	89	NA
n = 70	0	505	NA

2015			
Average state-space size (per 1000)			
	OS = 0.8	OS = 0.6	OS = 0.4
n = 10	0	0	1
n = 20	0	4	55
n = 30	2	49	1560
n = 40	8	534	47073
n = 50	27	4346	526020
n = 60	92	42279	NA
n = 70	287	216028	NA

No matter how fast my procedure is, I'll always be limited by memory constraints!

Think **Homer**!
Think! How can we reduce memory constraints?

What if I **relax the state space**? What if I only keep track of the set of finished activities?

New CTMC

- We Introduce a new CTMC where states are defined by the set of finished activities

$\Rightarrow$up to 2^n states (instead of 3^n states)

- We no longer use UDCs

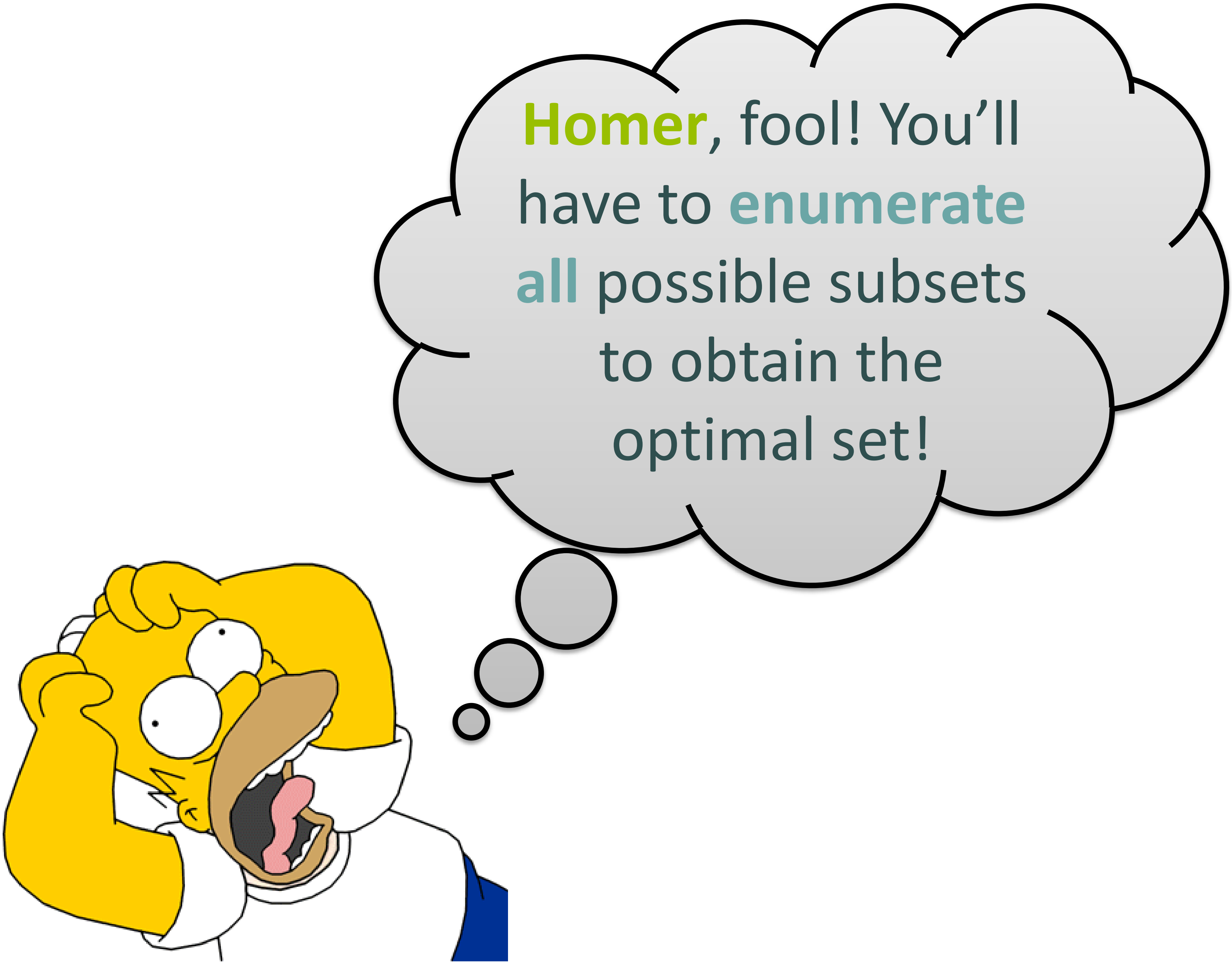
$\Rightarrow$no UDC network, no sorting, no tertiary values…

- Our procedure only generates feasible states in binary order => binary search is used to quickly retrieve states
- Significantly reduces CPU times & memory requirements!

This almost sounds too good to be true! Where is the catch?


If we only keep track of the finished activities, we do not know which activities are idle/ongoing!

We do know, however, what activities are **eligible** to start => we can determine the **optimal set of ongoing activities**


Homer, fool! You'll have to enumerate all possible subsets to obtain the optimal set!

On the other hand, we had to do that anyway in the old approach as well => perhaps it is **not as bad** as it sounds!

Even better: we can devise **heuristics** to determine a ***"good"*** set of ongoing activities!

But, if an activity is selected as a member of the set of ongoing activities in one state, does this mean that it is also selected in the next state? In this approach, it is possible that **activities are preempted**!

True! However, in reality preemption is a **realistic assumption** & even if preemption is not allowed, you have a nice **lower bound**!

Enough! Can we finally see some results!

2015 VS 2016 (new CTMC)
Average CPU time (sec)

2015			
Average CPU time (s)			
	OS = 0.8	OS = 0.6	OS = 0.4
n = 10	0	0	0
n = 20	0	0	0
n = 30	0	0	2
n = 40	0	1	92
n = 50	0	6	1048
n = 60	0	89	NA
n = 70	0	505	NA

2016			
Average CPU time (s)			
	OS = 0.8	OS = 0.6	OS = 0.4
n = 10	0	0	0
n = 20	0	0	0
n = 30	0	0	0
n = 40	0	0	7
n = 50	0	1	82
n = 60	0	6	NA
n = 70	0	34	NA

On average, we improve computation times by a factor of 13!

2015 VS 2016 (new CTMC)
Average number of states (per 1000)

2015			
Average state-space size (per 1000)			
	OS = 0.8	OS = 0.6	OS = 0.4
n = 10	0	0	1
n = 20	0	4	55
n = 30	2	49	1560
n = 40	8	534	47073
n = 50	27	4346	526020
n = 60	92	42279	NA
n = 70	287	216028	NA

2016			
Average state-space size (per 1000)			
	OS = 0.8	OS = 0.6	OS = 0.4
n = 10	0	0	0
n = 20	0	0	2
n = 30	0	2	17
n = 40	1	9	172
n = 50	2	40	1055
n = 60	4	175	NA
n = 70	8	593	NA

On average, we reduce memory requirements by a factor of 403!

New CTMC
Instances solved & CPU times

2016			
Instances solved			
	OS = 0.8	OS = 0.6	OS = 0.4
n = 10	30	30	30
n = 20	30	30	30
n = 30	30	30	30
n = 40	30	30	30
n = 50	30	30	30
n = 60	30	30	30
n = 70	30	30	30

We are able to solve way more instances!

New CTMC
Instances solved & CPU times

2016			
Instances solved			
	OS = 0.8	OS = 0.6	OS = 0.4
n = 10	30	30	30
n = 20	30	30	30
n = 30	30	30	30
n = 40	30	30	30
n = 50	30	30	30
n = 60	30	30	30
n = 70	30	30	30

2016			
Average CPU time (s)			
	OS = 0.8	OS = 0.6	OS = 0.4
n = 10	0	0	0
n = 20	0	0	0
n = 30	0	0	0
n = 40	0	0	22
n = 50	0	1	476
n = 60	0	11	16869
n = 70	0	99	263012

CPU times have become the new bottleneck

We use the new CTMC to tackle the SNPV and the PSRCPSP

Creemers (under review)

SNPV

- When solving the **SNPV**, activities are <u>NOT</u> preempted!
- **Why preempt**? To postpone a cash outflow as new information becomes available on the progress of the ongoing activities (e.g., an activity takes longer than expected => we can postpone another activity/cash flow).
- In Markovian PERT networks, activities have exponentially distributed durations
- The exponential distribution is **memoryless**

$\Rightarrow$ No new information becomes available on the progress of activities!

$\Rightarrow$ It doesn’t make sense to preempt activities!

- This finding is also confirmed in all our experiments

Creemers (also under review)

PSRCPSP

- **RCPSP** => **PSPLIB** instances
- We solve all instances of **J30** & **J60** with ease
- We solve 196 instances of **J90** & 10 of **J120**
- **Why preempt**? To avoid a **lockdown** of a resource (e.g., a resource is captured by an activity that takes longer than expected)
- For the deterministic RCPSP, the benefit of preemption is limited
- For the PSRCPSP the benefit of preemption is significant & increases with the size/complexity of the network!

Conclusion

- New CTMC that only keeps track of finished activities
- Significantly reduces memory requirements when compared with CTMC of Kulkarni & Adlakha
- Only “drawback” is that it allows activities to be preempted
- There is no preemption when solving the SNPV if activity durations are exponential

?