I initially intend to do options 1), but I still can change to option 2) if I notice it's better.

Ideas for programming NEAT-Gammon:

- Programming Language:
    - http://www.cs.ucf.edu/~kstanley/neat.html
    - 1) Java
        - NEAT:
            - http://nn.cs.utexas.edu/?neat-java
            - http://anji.sourceforge.net/
            - http://neat4j.sourceforge.net/
        - Bases source codes for Backgammon:
            - http://blog.alirabiee.com/?p=674
            - https://github.com/backgammon-java-ai/backg_java2012
            - http://jgam.sourceforge.net/
    - 2) C++
        - NEAT:
            - http://nn.cs.utexas.edu/?neat_original
            - http://nn.cs.utexas.edu/?neat-c
            - http://nn.cs.utexas.edu/?windowsneat
            - http://nn.cs.utexas.edu/?rtNEAT
        - Bases source codes for Backgammon:
            - http://www.gnubg.org/
            - http://www.sourcecodedownloads.com/342817/
    - Useful website: http://www.bkgm.com/rgb/rgb.cgi?view+593

- Comparable with:
    - 1) the first version of TD-Gammon (initially)
    - 2) TD-Gammon 1.0 (if I have enough time, I will improve the algorithm with features)
    - 3) TD-Gammon 2.1 (if I have enough time, I will improve the algorithm with look-ahead)

- Input: **I have a question here: What would be the best way to enforce the game rules to the network?**
    - 1) raw data, no features:
        - number of own and opponent's checkers in each of the 24 positions, codified as: zero, one, two, three or more.
        - Number of own checkers at the bar
        - Number of opponent checkers at the bar
        - Number of own checkers off board
        - Number of opponent checkers off board
    - 2) all valid moves

- Output: selected move
    - 1) two outputs, one for the piece that will be moved and another for the dice used
    - 2) one output codified as an object "Move"

- no doubling cube

- reward scheme: win (1 point), gammon (2 points), lose (0 points)

- As far as I noticed, playing backgammon isn't a fractured problem, because besides being necessary to develop complex strategies, the optimal actions change continuously as the game is played.
  - In case it shows up being a fractured problem, I will try to workaround it using RBF-NEAT, Cascade-NEAT or SNAP-NEAT.

- Benchmark:
  - main: TD-Gammon compatible with NEAT-Gammon algorithm
  - secondary: Neurogammon, Sun's Gammontool, Pubeval, if available.

- Methodology:
  - It will be used competitive coevolution, based on the paper (Competitive Coevolution through Evolutionary Complexification, 2004).
  - Test the NEAT network for the bear-off case, depending upon the results:
    - If I got optimistic results: I will apply it to play the full game.
    - If I didn't get optimistic results: I will apply it to play the racing case, and just after it I will apply it to play the full game, following a methodology similar to Tesauro's in his 1992 paper.

- Parameter tuning: Initially I will use the default parameters for the network, and then I will tune them to try to get better results.

- Research goals:
  - compare performance of NEAT-Gammon and TD-Gammon
  - describe the methodology used and results in details
  - analyze the strategies and topologies developed by NEAT